SQL Server Reference Manual

Volume 1: Commands, Functions, and Topics

SYBASE SQL Server Release 10.0
Document ID: 32401-01-1000-03
Change Level: 2
Last Revised: June 17, 1994

Principal authors: Server Publications Group

Document ID: 32401-01-1000

This publication pertains to SYBASE SQL Server Release 10.0 of the SYBASE database management software and to any subsequent release until otherwise indicated in new editions or technical notes. Information in this document is subject to change without notice. The software described herein is furnished under a license agreement, and it may be used or copied only in accordance with the terms of the agreement.

Document-Back Guarantee

Sybase welcomes corrections and comments on its documents. If you mark typographical errors, formatting errors, errors of fact, or areas that need clarification in any Sybase user's manual and send copies of marked-up pages to us, we will send you a clean copy of the manual, absolutely free**.**

Send pages to the Publications Operations Department at the address below. Please include your Site ID number.

Sybase, Inc.
6475 Christie Avenue
Emeryville, CA 94608
USA

(510) 922-3500
Fax (510) 922-5340

Document Orders

Customers may purchase additional copies of any document or the right to make photocopies of documentation for their in-house use.

To order additional documents or photocopy rights, U.S. and Canadian customers should call Customer Fulfillment at (800) 685-8225, fax (617) 229-9845.

Customers in other countries with a U.S. license agreement may contact Customer Fulfillment via the fax number. All other international customers should contact their Sybase subsidiary or local distributor.

Upgrades are provided only at regularly scheduled software release dates.

©Copyright Sybase, Inc., 1989, 1994. All rights reserved.

No part of this publication may be reproduced, transmitted, or translated in any form or by any means, electronic, mechanical, manual, optical or otherwise, without prior written permission of Sybase, Inc.

Sybase Trademarks

SYBASE, the SYBASE logo, APT-FORMS, Data Workbench, DBA Companion, Deft, GainExposure, GainInsight, Gain*Momentum*, SA Companion, SQL Debug, SQL Solutions, SQR, Transact-SQL, and VQL are registered trademarks of Sybase, Inc. Adaptable Windowing Environment, ADA Workbench, Application Manager, Applications from Models, APT-Build, APT-Edit, APT-Execute, APT-Library, APT-Translator, APT Workbench, Build *Momentum*, Camelot, Client / Server Architecture for the Online Enterprise, Client / Server for the Real World, Configurator, Database Analyzer, DBA Companion Application Manager, DBA Companion Resource Manager, DB-Library, Deft Analyst, Deft Designer, Deft Educational, Deft Professional, Deft Trial, Developers Workbench, Easy SQR, Embedded SQL, Enterprise Builder, Enterprise Client / Server, Enterprise Meta Server, Enterprise Modeler, Enterprise *Momentum*, Gain, Insight, MAP, Maintenance Express, MethodSet, Movedb, Navigation Server, Net-Gateway, Net-Library, Object *Momentum*, OmniSQL Access Module, OmniSQL Gateway, OmniSQL Server, Open Client, Open Client / Server Interfaces, Open Gateway, Open Server, Open Solutions, Partnerships That Work, PC APT-Execute, PC DB-Net, PC Net Library, PostDoc, Replication Server, Replication Server Manager, Report-Execute, Report Workbench, Resource Manager, RW-Display Lib, RW-Library, Secure SQL Server, Secure SQL Toolset, SQL Code Checker, SQL Edit, SQL Edit / TPU, SQL Monitor, SQL Server, SQL Server / CFT, SQL Server / DBM, SQL Station, SQL Toolset, SQR Developers Kit, SQR Execute, SQR Toolset, SQR Workbench, SYBASE Client / Server Interfaces, SYBASE Gateways, Sybase *Momentum*, SYBASE SQL Lifecycle, Sybase Synergy Program, SYBASE Virtual Server Architecture, SYBASE User Workbench, System 10, Tabular Data Stream, The Enterprise Client / Server Company, and The Online Information Center are trademarks of Sybase, Inc.

All other company and product names used herein may be the trademarks or registered trademarks of their respective companies.

Restricted Rights Legend

Use, duplication or disclosure by the Government is subject to restrictions set forth in subparagraph (c)(1)(ii) of DFARS 52.227-7013 for the DOD and as set forth in FAR 52.227-19(a)-(d) for civilian agencies.

Sybase, Inc., 6475 Christie Avenue, Emeryville, CA 94608

Table of Contents

Preface

Audience xvii
How to Use this Book xvii
Related Documents xviii
Conventions Used in This Manual xix
If You Need Help xxii

1. Transact-SQL Commands

alter database 1-6
alter table 1-10
begin...end 1-19
begin transaction 1-21
break 1-23
checkpoint 1-25
close 1-27
commit 1-28
compute Clause 1-30
continue 1-39
create database 1-41
create default 1-46
create index 1-50
create procedure 1-58
create rule 1-68
create schema 1-72
create table 1-75
create trigger 1-94
create view 1-103
dbcc 1-110
deallocate cursor 1-116
declare 1-117
declare cursor 1-119
delete 1-125
disk init 1-130
disk mirror 1-134
disk refit 1-137

disk reinit 1-138
disk remirror 1-140
disk unmirror 1-143
drop database 1-146
drop default 1-148
drop index 1-150
drop procedure 1-152
drop rule 1-154
drop table 1-156
drop trigger 1-159
drop view 1-160
dump database 1-162
dump transaction 1-174
execute 1-187
fetch 1-192
goto Label 1-195
grant 1-196
group by and having Clauses 1-207
if...else 1-217
insert 1-219
kill 1-227
load database 1-230
load transaction 1-238
open 1-246
order by Clause 1-248
prepare transaction 1-252
print 1-253
raiserror 1-256
readtext 1-261
reconfigure 1-264
return 1-266
revoke 1-269
rollback 1-275
rollback trigger 1-277
save transaction 1-279
select 1-281
set 1-293
setuser 1-306
shutdown 1-308

truncate table 1-311
union Operator 1-313
update 1-317
update statistics 1-324
use 1-326
waitfor 1-327
where Clause 1-329
while 1-336
writetext 1-338

2. Transact-SQL Functions

Aggregate Functions 2-2
Datatype Conversion Functions 2-9
Date Functions 2-19
Mathematical Functions 2-24
Row Aggregate Functions 2-29
String Functions 2-33
System Functions 2-40
text and image Functions 2-48

3. Transact-SQL Topics

Auditing 3-3
Batch Queries 3-6
Browse Mode 3-8
Comments 3-10
Control-of-Flow Language 3-12
Cursors 3-14
Datatypes 3-25
Disk Mirroring 3-47
Expressions 3-52
Identifiers 3-61
IDENTITY Columns 3-67
Joins 3-79
Login Management 3-85
Null Values 3-88
Parameters 3-96
Roles 3-99
Search Conditions 3-105

Subqueries 3-110
Temporary Tables 3-116
text and image Datatypes. 3-120
Transactions 3-125
Variables (Local and Global) 3-143
Wildcard Characters 3-150

A. Reserved Words

Transact-SQL Reserved Words A-1
APT-SQL Keywords A-3
SQL92 Keywords A-4
Potential SQL92 Reserved Words A-6

B. The System Tables

Introduction B-1
sysalternates B-5
sysauditoptions B-6
sysaudits B-8
syscharsets B-11
syscolumns B-13
syscomments B-15
sysconfigures B-16
sysconstraints B-18
syscurconfigs B-19
sysdatabases B-20
sysdepends B-22
sysdevices B-23
sysengines B-25
sysindexes B-26
syskeys B-29
syslanguages B-31
syslocks B-33
sysloginroles B-35
syslogins B-36
syslogs B-38
sysmessages B-39
sysobjects B-40
sysprocedures B-42

sysprocesses B-43
sysprotects B-45
sysreferences B-47
sysremotelogins B-49
sysroles B-50
syssegments B-51
sysservers B-52
syssrvroles B-53
systhresholds B-54
systypes B-55
sysusages B-57
sysusermessages B-58
sysusers B-59

C. The *pubs2* Database

Tables in the *pubs2* Database C-1
Primary and Foreign Keys in *pubs2* C-18
Other Objects in *pubs2* C-19
Diagram of the *pubs2* Database C-20

D. SQLSTATE Codes and Messages

Warnings D-1
Exceptions D-2

Index

List of Figures

Figure 1-1: File naming convention for database dumps 1-170
Figure 1-2: Dumping several databases to the same volume..... 1-172
Figure 1-3: File naming convention for transaction log dumps..... 1-184
Figure 1-4: Dumping three transaction logs to a single volume..... 1-185
Figure 3-1: Nesting transaction statements..... 3-134

List of Tables

Table 1: Syntax statement conventions xix
Table 2: Types of expressions used in syntax statements xxi
Table 1-1: Transact-SQL commands 1-1
Table 1-2: Information stored about referential integrity constraints 1-16
Table 1-3: Row aggregate functions used with the *compute* clause 1-30
Table 1-4: Relationship between nulls and column defaults 1-48
Table 1-5: Duplicate row options 1-53
Table 1-6: Index options 1-53
Table 1-7: Rule binding precedence 1-69
Table 1-8: Variable-length datatypes used to store nulls 1-83
Table 1-9: Methods of integrity enforcement 1-85
Table 1-10: Information stored about cross-database referential integrity constraints 1-90
Table 1-11: Information stored about referential integrity constraints 1-157
Table 1-12: Commands used to back up databases and logs 1-167
Table 1-13: Commands used to back up logs and databases 1-179
Table 1-14: *@@sqlstatus* values 1-193
Table 1-15: Object access permissions 1-197
Table 1-16: Command and object permissions 1-200
Table 1-17: Status values reported by *sp_who* 1-228
Table 1-18: Commands used to restore databases from dumps 1-233
Table 1-19: Commands used to restore databases from dumps 1-241
Table 1-20: Effect of sort order choices 1-250
Table 1-21: SQL Server error return values 1-268
Table 1-22: Object access permissions 1-270
Table 1-23: Results of using aggregates with *group by* 1-285
Table 1-24: Permissions required for *update* and *delete* 1-294
Table 1-25: *set* options for entry level SQL92 compliance 1-305
Table 1-26: Comparing datatypes in union operations 1-315
Table 1-27: Comparison operators 1-330
Table 1-28: Arithmetic operators 1-331
Table 1-29: Bitwise operators 1-331
Table 1-30: Wildcard operators 1-333
Table 2-1: Transact-SQL functions 2-1
Table 2-2: Aggregate functions 2-2
Table 2-3: Datatype conversion functions 2-9
Table 2-4: Display formats for date / time information 2-10
Table 2-5: Explicit, implicit, and unsupported datatype conversions 2-12
Table 2-6: Date functions 2-19

Table 2-7: Date parts and their values 2-20
Table 2-8: Arguments used in mathematical functions....2-24
Table 2-9: Mathematical functions 2-25
Table 2-10: Row aggregate functions 2-29
Table 2-11: Arguments used in string functions 2-33
Table 2-12: Function names, arguments, and results....2-34
Table 2-13: Results of the *right*, *str*, and *stuff* functions 2-37
Table 2-14: Arguments used in system functions 2-40
Table 2-15: System functions, arguments, and results 2-41
Table 2-16: *text* and *image* functions, arguments, and results....2-48
Table 3-1: Transact-SQL topics....3-1
Table 3-2: System procedures used to manage auditing options 3-4
Table 3-3: Control-of-flow keywords....3-12
Table 3-4: *@@sqlstatus* values 3-21
Table 3-5: Range and storage size for SQL Server system datatypes....3-25
Table 3-6: Integer datatypes 3-27
Table 3-7: Approximate numeric datatypes....3-28
Table 3-8: Character datatypes....3-30
Table 3-9: Money datatypes 3-34
Table 3-10: Precision and scale after arithmetic operations....3-44
Table 3-11: Automatic conversion of fixed-length to variable-length datatypes 3-45
Table 3-12: Arithmetic operators 3-53
Table 3-13: Truth tables for bitwise operations....3-54
Table 3-14: Examples of bitwise operations....3-55
Table 3-15: Comparison operators 3-56
Table 3-16: Wildcards used with *like*....3-57
Table 3-17: Truth tables for logical expressions....3-58
Table 3-18: Comparison operators 3-80
Table 3-19: System procedures for managing SQL Server login accounts 3-85
Table 3-20: Conversion of fixed-length to variable-length datatypes....3-89
Table 3-21: Column definition and null defaults 3-92
Table 3-22: Roles required for SQL commands 3-101
Table 3-23: Roles required for system procedures 3-102
Table 3-24: Wildcards used in match strings 3-106
Table 3-25: Arithmetic operators 3-111
Table 3-26: Bitwise operators 3-111
Table 3-27: Comparison operators 3-112
Table 3-28: Storage of *text* and *image* data....3-121
Table 3-29: *@@transtate* values 3-126
Table 3-30: DDL commands allowed in transactions....3-129
Table 3-31: DDL commands not allowed in transactions 3-129

Table 3-32: How rollbacks affect processing 3-135
Table 3-33: How rollbacks from errors affect processing 3-139
Table 3-34: How rollbacks from duplicate key or rules errors affect processing 3-141
Table 3-35: *@@sqlstatus* Values 3-147
Table 3-36: *@@transtate* Values 3-148
Table 3-37: Using square brackets to search for wildcards 3-153
Table 3-38: Using the *escape* clause 3-154
Table A-1: Transact-SQL reserved words A-1
Table A-2: APT-SQL keywords A-3
Table A-3: SQL92 keywords that are not Transact-SQL reserved words A-4
Table A-4: Potential reserved words A-6
Table B-1: System tables that occur in all databases B-1
Table B-2: System tables that occur in the *master* database only B-2
Table B-3: System tables that occur in the *sybsecurity* database only B-3
Table B-4: Columns in the *sysalternates* table B-5
Table B-5: Columns in the *sysauditoptions* table B-6
Table B-6: Audit option values and descriptions B-6
Table B-7: Columns in the *sysaudits* table B-8
Table B-8: Contents of *event* and *extrainfo* columnss of *sysaudits* B-9
Table B-9: Columns in the *syscharsets* table B-11
Table B-10: Columns in the *syscolumns* table B-13
Table B-11: Columns in the *syscomments* table B-15
Table B-12: Columns in the *sysconfigures* table B-16
Table B-13: Contents of *sysconfigures* B-16
Table B-14: Columns in the *sysconstraints* table B-18
Table B-15: Columns in the *syscurconfigs* table B-19
Table B-16: Columns in the *sysdatabases* table B-20
Table B-17: *status* control bits in the *sysdatabases* table B-21
Table B-18: *status2* control bits in the *sysdatabases* table B-21
Table B-19: Columns in the *sysdepends* table B-22
Table B-20: Columns in the *sysdevices* table B-23
Table B-21: *status* control bits in the *sysdevices* table B-23
Table B-22: Columns in the *sysengines* table B-25
Table B-23: Columns in the *sysindexes* table B-26
Table B-24: *status2* control bits in the *sysindexes* table B-27
Table B-25: *status* control bits in the *sysindexes* table B-28
Table B-26: Columns in the *syskeys* table B-29
Table B-27: Columns in the *syslanguages* table B-31
Table B-28: Columns in the *syslocks* table B-33
Table B-29: *type* control bit in the *syslocks* table B-33
Table B-30: Columns in the *sysloginroles table* B-35

Table B-31: Columns in the *syslogins* table B-36
Table B-32: *status* control bits in the *syslogins* table B-37
Table B-33: Columns in the *syslogs* table B-38
Table B-34: Columns in the *sysmessages* table B-39
Table B-35: Columns in the *sysobjects* table B-40
Table B-36: *systat2* control bits in the *sysobjects* table B-41
Table B-37: Columns in the *sysprocedures* table B-42
Table B-38: *type* control bits in the *sysprocedures* table B-42
Table B-39: Columns in the *sysprocesses* table B-43
Table B-40: Columns in the *sysprotects* table B-45
Table B-41: Columns in the *sysreferences* table B-47
Table B-42: Columns in the *sysremotelogins* table B-49
Table B-43: Columns in the *sysroles* table B-50
Table B-44: Columns in the *syssegments* table B-51
Table B-45: Columns in the *sysservers* table B-52
Table B-46: Columns in the *syssrvroles* table B-53
Table B-47: Columns in the *systhresholds* table B-54
Table B-48: Columns in the *systypes* table B-55
Table B-49: Datatype names, hierarchy, types, and usertypes B-56
Table B-50: Columns in the *sysusages* table B-57
Table B-51: Columns in the *sysusermessages* table B-58
Table B-52: Columns in the *sysusers* table B-59
Table D-1: Warnings D-1
Table D-2: Cardinality violations D-2
Table D-3: Data exceptions D-3
Table D-4: Integrity constraint violations D-4
Table D-5: Invalid cursor states D-5
Table D-6: Syntax errors and access rule violations D-6
Table D-7: Transaction rollbacks D-7
Table D-8: With check option violations D-7

Preface

The *SQL Server Reference Manual* is a two-volume guide to SYBASE SQL Server™ and the Transact-SQL® language. This volume includes information about Transact-SQL commands, built-in functions, and topics of general interest to Transact-SQL users. Volume 2 contains information about system procedures and catalog stored procedures.

Audience

This manual is intended as a reference tool for Transact-SQL users of all levels. It provides basic syntax and usage information for every command, function, system procedure, and catalog stored procedure.

How to Use this Book

This manual consists of the following chapters:

- Chapter 1, "Transact-SQL Commands," contains reference information for every Transact-SQL command. Particularly complex commands, such as **select**, are divided into subsections. For example, there are reference pages on the **compute** clause and on the **group by** and **having** clauses of the **select** command.
- Chapter 2, "Transact-SQL Functions," contains reference information for the SQL Server aggregate functions, datatype conversion functions, date functions, mathematical functions, row aggregate functions, string functions, system functions, and text/image functions.
- Chapter 3, "Transact-SQL Topics," contains information about topics of general interest. Each topic describes the commands, system procedures, and functions that provide a particular type of functionality in Transact-SQL.
- Appendix A, "Reserved Words" lists Transact-SQL, APT-SQL, and SQL2 reserved words.
- Appendix B, "The System Tables" lists each system table and the definition of all columns in each table.

- Appendix C, ‘‘The pubs2 Database,’’ describes the *pubs2* database in detail. It also includes an entity relationship diagram of all the tables in *pubs2*.
- Appendix D, ‘‘SQLSTATE Codes and Messages,’’ describes the SQLSTATE codes and associated messages that are generated when a warning or exception condition is detected.
- The index contains entries for both Volume 1 and Volume 2 of the *SQL Server Reference Manual*.

Related Documents

Other manuals that you may find useful are:

- *What’s New in SYBASE SQL Server Release 10.0?*, which describes the new software and documentation features of this release.
- *Transact-SQL User’s Guide*, which documents Transact-SQL, Sybase’s enhanced version of the relational database language. It serves as a textbook for beginning users of the SQL Server.
- *System Administration Guide*, which contains in-depth information about SQL Server system administration issues.
- *System Administration Guide Supplement* for your operating system, which documents operating-system specific system administration tasks.
- *Open Client DB-Library Reference Manual*, a collection of manual pages and code samples for the SQL Server interface library, Open Client DB-Library.
- The *SQL Server Installation Guide*, which describes the installation procedures for SQL Server.
- *SQL Server Utility Programs* for your operating system, which describes utility programs that are invoked directly from the operating system.
- *Master Index for Server Publications*, which combines the indexes of the *SQL Server Reference Manual, Transact-SQL User’s Guide*, and *System Administration Guide*.

Conventions Used in This Manual

Formatting SQL Statements

SQL is a free-form language: there are no rules about the number of words you can put on a line, or where you must break a line. However, for readability, all examples and syntax statements in this manual are formatted so that each clause of a statement begins on a new line. Clauses that have more than one part extend to additional lines, which are indented.

SQL Syntax Conventions

The conventions for syntax statements in this manual are as follows:

Key	Definition
command	Command names, command option names, utility names, utility flags, and other keywords are in **`bold Courier`** in syntax statements, and in **bold Helvetica** in paragraph text.
variable	Variables, or words that stand for values that you fill in, are in italics.
{ }	Curly braces indicate that you choose at least one of the enclosed options. Do not include braces in your option.
[ ]	Brackets mean choosing one or more of the enclosed options is optional. Do not include brackets in your option.
( )	Parentheses are to be typed as part of the command.
|	The vertical bar means you may select only one of the options shown.
,	The comma means you may choose as many of the options shown as you like, separating your choices with commas to be typed as part of the command.

Table 1: Syntax statement conventions

- Syntax statements (displaying the syntax and all options for a command) are printed like this:

  `sp_dropdevice [`*`device_name`*`]`

or, for a command with more options:

```
select column_name
    from table_name
    where search_conditions
```

In syntax statements, keywords (commands) are in normal font and identifiers are in lowercase: normal font for keywords, italics for user-supplied words.

- Examples showing the use of Transact-SQL commands are printed like this:

```
select * from publishers
```

- Examples of output from the computer are printed like this:

```
pub_id  pub_name                city         state
------- -------------------     ------------ -----
0736    New Age Books           Boston       MA
0877    Binnet & Hardley        Washington   DC
1389    Algodata Infosystems    Berkeley     CA

(3 rows affected)
```

Case

You can disregard case when you type keywords:

SELECT is the same as **Select** is the same as **select**

SQL Server's sensitivity to the case (upper or lower) of database objects, such as table names, and data depends on the sort order installed on your Server. Case sensitivity can be changed for single-byte character sets by reconfiguring SQL Server's sort order. (See the *System Administration Guide* for more information.)

Obligatory Options {You Must Choose At Least One}

- **Curly Braces and Vertical Bars**: Choose **one and only one** option.

```
{die_on_your_feet | live_on_your_knees |
live_on_your_feet}
```

- **Curly Braces and Commas**: Choose one or more options. If you choose more than one, separate your choices with commas.

```
{cash, check, credit}
```

Optional Options [You Don't Have to Choose Any]

- **One Item in Square Brackets**: You don't have to choose it.

  ```
  [anchovies]
  ```

- **Square Brackets and Vertical Bars:** Choose **none or only one**.

  ```
  [beans | rice | sweet_potatoes]
  ```

- **Square Brackets and Commas**: Choose **none, one, or more than one** option. If you choose more than one, separate your choices with commas.

  ```
  [extra_cheese, avocados, sour_cream]
  ```

Ellipsis: Do It Again (and Again)...

An ellipsis (...) means that you can **repeat** the last unit as many times as you like. In this syntax statement, **buy** is a required keyword:

```
buy thing = price [cash | check | credit]
    [, thing = price [cash | check | credit]]...
```

You must buy at least one thing and give its price. You may choose a method of payment: one of the items enclosed in square brackets. You may also choose to buy additional things: as many of them as you like. For each thing you buy, give its name, its price, and (optionally) a method of payment.

Expressions

Several different types of expressions are used in SQL Server syntax statements.

Usage	Definition
expression	Can include constants, literals, functions, column identifiers, variables or parameters
logical expression	An expression that returns TRUE, FALSE or UNKNOWN
constant expression	An expression that always returns the same value, such as "5+3" or "ABCDE"
float_expr	Any floating-point expression or expression that implicitly converts to a floating value
integer_expr	Any integer expression, or an expression that implicitly converts to an integer value

Table 2: Types of expressions used in syntax statements

Usage	Definition
numeric_expr	Any numeric expression that returns a single value
char_expr	Any expression that returns a single character-type value
binary_expression	An expression that returns a single *binary* or *varbinary* value

Table 2: Types of expressions used in syntax statements

If You Need Help

Help with your SYBASE software is available in the form of documentation and the Technical Support Center.

Each SYBASE installation has a designated person who may contact Technical Support. If you cannot resolve your problem using the manuals, ask the designated person at your site to contact SYBASE Technical Support.

Commands

1 Transact-SQL Commands

This chapter describes commands, clauses, and other elements used to construct a Transact-SQL statement. Table 1-1 lists the elements discussed in this chapter and describes the function of each element.

Command	Description
alter database	Increases the amount of space allocated to a database.
alter table	Adds new columns and constraints, changes constraints, or drops constraints on an existing table.
begin...end	Encloses a series of SQL statements so that control-of-flow language, such as **if ... else**, can affect the performance of the whole group.
begin transaction	Marks the starting point of a user-defined transaction.
break	Causes an exit from a **while** loop. **break** is often activated by an **if** test.
checkpoint	Writes all "dirty" pages (pages that have been updated since they were last written) to the database device.
close	Deactivates a cursor.
commit	Marks the ending point of a user-defined transaction.
compute clause	Generates summary values that appear as additional rows in the query results.
continue	Causes the **while** loop to restart. **continue** is often activated by an **if** test**.**
create database	Creates a new database.
create default	Specifies a value to insert in a column (or in all columns of a user-defined datatype) if no value is explicitly supplied at insert time.
create index	Creates an index on one or more columns in a table.
create procedure	Creates a stored procedure that can take one or more user-supplied parameters.
create rule	Specifies the domain of acceptable values for a particular column or for any column of a user-defined datatype.
create schema	Creates a new collection of tables, views and permissions for a database user.

Table 1-1: Transact-SQL commands

Command	Description
create table	Creates new tables and optional integrity constraints.
create trigger	Creates a trigger, a type of stored procedure often used for enforcing integrity constraints. A trigger executes automatically when a user attempts a specified data modification statement on a specified table.
create view	Creates a view, which is an alternative way of looking at the data in one or more tables.
dbcc	Database Consistency Checker (**dbcc**) checks the logical and physical consistency of a database. **dbcc** should be used regularly as a periodic check, or if damage is suspected.
deallocate cursor	Makes a cursor inaccessible and releases all memory resources committed to that cursor.
declare	Declares the name and type of local variables for a batch or procedure. Local variables are assigned values with a **select** statement.
declare cursor	Defines a cursor.
delete	Removes rows from a table.
disk init	Makes a physical device or file usable by SQL Server. (It is not necessary to use **disk init** on the master device, which is initialized by the **sybinit** installation program.)
disk mirror	Creates a software mirror that immediately takes over when the primary device fails. You can mirror the master device, devices that store data, and devices that store transaction logs; you cannot mirror dump devices.
disk refit	Rebuilds the *master* database's *sysusages* and *sysdatabases* system tables from information contained in *sysdevices*. Use **disk refit** after **disk reinit** as part of the procedure to restore the *master* database.
disk reinit	Rebuilds the *master* database's *sysdevices* system table. Use **disk reinit** as part of the procedure to restore the *master* database.
disk remirror	Reenables disk mirroring after it is stopped by failure of a mirrored device or temporarily disabled by the **disk unmirror** command.
disk unmirror	Disables either the original device or its mirror, allowing hardware maintenance or the changing of a hardware device.
drop database	Removes one or more databases from a SQL Server.

Table 1-1: Transact-SQL commands (continued)

Command	Description
drop default	Removes a user-defined default.
drop index	Removes an index from a table in the current database.
drop procedure	Removes user-defined stored procedures.
drop rule	Removes a user-defined rule.
drop table	Removes a table definition and all of its data, indexes, triggers, and permission specifications from the database.
drop trigger	Removes a trigger.
drop view	Removes one or more views from the current database.
dump database	Makes a backup copy of the entire database, including the transaction log, in a form that can be read in with **load database.**
dump transaction	Makes a copy of a transaction log and removes the inactive portion.
	no_truncate copies the log without truncating it. Use when the database is inaccessible after device failure.
	truncate_only truncates the log without copying it. Use for databases without a separate log segment and databases for which you do not need an up-to-date dump.
	no_log truncates a log without copying it and without logging the event. **Use only as a last resort**, when your usual method of dumping the transaction log fails because of insufficient log space**.**
execute	Runs a system procedure or a user-defined stored procedure.
fetch	Returns a row or a set of rows from a cursor result set.
goto label	Branches to a user-defined label.
grant	Assigns permissions to users.
group by and **having** clauses	Used in **select** statements to divide a table into groups.
if...else	Imposes conditions on the execution of a SQL statement. The statement following an **if** keyword and its condition is executed if the condition is satisfied (when the logical expression returns “true”). The optional **else** keyword introduces an alternate SQL statement that executes when the **if** condition is not satisfied (when the logical expression returns “false”).
insert	Adds new rows to a table or view.

Table 1-1: Transact-SQL commands (continued)

Command	Description
kill	Kills a process.
load database	Loads a backup copy of a user database, including its transaction log. The **listonly** and **headeronly** options display information about the dump files without loading them.
load transaction	Loads a backup copy of the transaction log. The **listonly** and **headeronly** options display information about the dump files without loading them.
open	Opens a cursor for processing.
order by clause	Returns query results in the specified column(s) in sorted order.
prepare transaction	Used by DB-Library in a two-phase commit application to see if a server is prepared to commit a transaction.
print	Prints a user-defined message on the user's screen.
raiserror	Prints a user-defined error message on the user's screen and sets a system flag to record that an error condition has occurred.
readtext	Reads *text* and *image* values, starting from a specified offset and reading a specified number of bytes or characters.
reconfigure	Sets configuration variables that control various aspects of SQL Server's memory allocation, performance, and options. Used with the system procedure **sp_configure.**
return	Exits from a batch or procedure unconditionally, optionally providing a return status. Statements following **return** are not executed.
revoke	Revokes permissions from users.
rollback	Rolls a user-defined transaction back to the last savepoint inside the transaction or to the beginning of the transaction.
rollback trigger	Rolls back the work done in a trigger, including the update that caused the trigger to fire, and issues an optional **raiserror** statement.
save transaction	Sets a savepoint within a transaction.
select	Retrieves rows from database objects.
set	Sets SQL Server query-processing options for the duration of the user's work session. Can be used to set some options inside a trigger or stored procedure.

Table 1-1: Transact-SQL commands (continued)

Command	Description
setuser	Allows a Database Owner to impersonate another user.
shutdown	Shuts down SQL Server or a Backup Server. This command can only be issued by a System Administrator.
truncate table	Removes all rows from a table.
union operator	Returns a single result set that combines the results of two or more queries. Duplicate rows are eliminated from the result set unless the **all** keyword is specified.
update	Changes data in existing rows, either by adding data or by modifying existing data.
update statistics	Updates information about the distribution of key values in specified indexes.
use	Specifies the database with which you want to work.
waitfor	Specifies a specific time, a time interval, or an event for the execution of a statement block, stored procedure, or transaction.
where clause	Sets the search conditions in a **select, insert, update, or delete** statement. (Joins and subqueries are specified in the search conditions: see the "Joins" and "Subqueries" sections for full details.)
while	Sets a condition for the repeated execution of a statement or statement block. The statement(s) are executed repeatedly as long as the specified condition is true.
writetext	Permits non-logged, interactive updating of an existing **text** or **image** column**.**

Table 1-1: Transact-SQL commands (continued)

alter database

Function

Increases the amount of space allocated to a database.

Syntax

```
alter database database_name
   [on {default | database_device } [= size]
       [, database_device [= size]]...]
   [log on { default | database_device } [ = size ]
       [ , database_device [= size]]...]
   [with override]
   [for load]
```

Keywords and Options

database_name – is the name of the database. The database name can be a literal, a variable, or a stored procedure parameter.

on – indicates that you want to specify a size and / or location for the database extension. If you have your log and data on separate device fragments, use this clause for the data device, and the **log on** clause for the log device.

default – indicates that **alter database** can put the database extension on any default database device(s) (as shown by **sp_helpdevice**). To specify a size for the database extension without specifying the exact location, use this command:

```
on default = size
```

To change a database device's status to default, use the system procedure **sp_diskdefault**.

database_device – is the name of the database device on which you want to locate the database extension. A database can occupy more than one database device with different amounts of space on each.

size – is the amount of space, in megabytes, allocated to the database extension. The minimum extension is 1 MB (512 2K pages). The default value is 1 MB. Legal values for the *size* parameter range from 1 to 2048 MB.

log on– indicates that you want to specify additional space for the database's transaction logs. See the information above about

default, *database_device*, and *size*. The **log on** clause uses the same defaults as the **on** clause.

with override – forces SQL Server to accept your device specifications, even if they mix data and transaction logs on the same device, thereby endangering up-to-the-minute recoverability for your database. If you attempt to mix log and data on the same device without this clause, the **alter database** command fails. If you mix log and data, and use **with override**, you are warned, but the command succeeds.

for load – is used only after **create database for load**, when you must recreate the space allocations and segment usage of the database being loaded from a dump.

Examples

1.
```
alter database mydb
```

   Adds 1 MB to the database *mydb* on a default database device.

2.
```
alter database pubs2
on newdata = 3
```

   Adds 3 MB to the space allocated for the *pubs2* database on the database device named *newdata*.

3.
```
alter database production
    on userdata1 = 10
    log on logdev = 2
```

   Adds 10 MB of space for data on *userdata1* and 2 MB for the log on *logdev*.

Comments

Restrictions

- You must be using the *master* database, or executing a stored procedure in the *master* database, to use **alter database**.
- If SQL Server cannot allocate the requested space, it comes as close as possible per device and prints a message telling how much space has been allocated on each database device.
- You can only expand the *master* database on the master device, *d_master*. An attempt to use **alter database** to expand the *master* database to any other database device results in an error message.

Here is an example of the correct statement for modifying the *master* database on the master device:

```
alter database master on d_master = 1
```

- The maximum number of device fragments for any database is 128. Each time you allocate space on a database device with **create database** or **alter database**, that allocation represents a device fragment, and the allocation is entered as a row in *sysusages*.
- If you use **alter database** on a database that is in the process of being dumped, the **alter database** command cannot complete until the dump finishes. SQL Server locks the in-memory map of database space use during a dump. If you issue an **alter database** command while this in-memory map is locked, SQL Server updates the map from the disk after the dump completes. If you interrupt **alter database**, you are instructed to run **sp_dbremap**. If you fail to run **sp_dbremap**, the space you added will not become available to SQL Server until the next reboot.

Backing Up master After Allocating More Space

- Back up the *master* database with the **dump database** command after each use of **alter database**. This makes recovery easier and safer in case *master* becomes damaged. (If you use **alter database** and fail to back up *master*, you may be able to recover the changes with **disk refit**.)

Placing the Log on a Separate Device

- To increase the amount of storage space allocated for the transaction log when you have used the **log on** extension to **create database**, give the name of the log's device in the **log on** clause when you issue the **alter database** command.
- If you did not use the **log on** extension of **create database** to place your logs on a separate device, you may not be able to recover fully in case of a hard disk crash. However, you can extend your logs in this case by using **alter database** with the **log on** clause, and then **sp_logdevice**.

Getting Help on Space Usage

- To see the names, sizes, and usage of device fragments already in use by a database, execute **sp_helpdb *dbname***.
- To see how much space the current database is using, execute **sp_spaceused**.

The System and Default Segments

- The *system* and *default* segments are mapped to each new database device included in the **on** clause of an **alter database** command. To unmap these segments, use **sp_dropsegment**.
- When you use **alter database** (without **override)** to extend a database on a device already in use by that database, the segments mapped to that device are also extended. If you use the **override** clause, all device fragments named in the **on** clause become system / default segments, and all device fragments named in the **log on** clause become log segments.

Using alter database *to Awaken Sleeping Processes*

- If user processes are suspended due to reaching a last-chance threshold on a log segment, adding space to the log segment awakens the processes.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

Permissions

alter database permission defaults to the Database Owner. System Administrators can also alter databases.

See Also

Commands	**create database, drop database, load database**
System procedures	**sp_addsegment, sp_diskdefault, sp_dropsegment, sp_helpdb, sp_helpsegment, sp_logdevice, sp_renamedb, sp_spaceused**

alter table

Function

Adds new columns and constraints, changes constraints, or drops constraints on an existing table.

Syntax

```
alter table [database.[owner].]table_name
   {add column_name datatype
       [default {constant_expression | user | null}]
       {[{identity | null}]
       | [[constraint constraint_name]
           {{unique | primary key}
               [clustered | nonclustered]
               [with fillfactor = x] [on segment_name]
           | references [[database.]owner.]ref_table
               [(ref_column)]
           | check (search_condition)}]}...
       {[, next_column]}...

   | add {[constraint constraint_name]
       {unique | primary key}
           [clustered | nonclustered]
           (column_name [{, column_name}...])
           [with fillfactor = x] [on segment_name]
       | foreign key (column_name [{, column_name}...])
           references [[database.]owner.]ref_table
               [(ref_column [{, ref_column}...])]
       | check (search_condition)}

   | drop constraint constraint_name

   | replace column_name
       default {constant_expression | user | null}}
```

Keywords and Options

table_name – is the name of the table you want to change.

add – specifies the name of the column or constraint you want to add to the table.

column_name – is the name of a column in that table.

datatype – is any of the system datatypes except *bit*, or any user-defined datatype except those based on *bit*.

default – specifies a default value for a column. If you have declared a default and the user does not provide a value for this column when inserting data, SQL Server inserts this value. The default can be a *constant_expression*, **user** (to insert the name of the user who is performing the insert), or **null** (to insert the null value).

constant_expression – is a constant expression to use as a default value for a column. It cannot include the name of any columns or other database objects, but you can include built-in functions that do not reference database objects. This default value must be compatible with the datatype of the column.

user | **null** – specifies that SQL Server should insert the user name or the null value as the default if the user does not supply a value. For **user**, the datatype of the column must be either *char(30)* or *varchar(30)*. For **null**, the column must allow null values.

identity – indicates that the column has the IDENTITY property. Each table in a database can have one IDENTITY column of type *numeric* and scale zero. IDENTITY columns are not updatable and do not allow nulls.

IDENTITY columns are used to store sequential numbers, such as invoice numbers or employee numbers, that are generated automatically by SQL Server. The value of the IDENTITY column uniquely identifies each row in a table.

null – specifies that SQL Server assigns a null value if a user does not provide a value during an insertion and no default exists.

constraint – introduces the name of an integrity constraint. This keyword and the *constraint_name* are optional.

constraint_name – is the name of the constraint. It must conform to the rules for identifiers and be unique in the database. If you do not specify the name for a table-level constraint, SQL Server generates a name as follows:

tabname_colname_objectid

where *tabname* is the first 10 characters of the table name, *colname* is the first 5 characters of the column name, and *objectid* is the object ID number for the constraint. If you do not specify the name for a unique or primary key constraint, SQL Server generates a name as follows:

tabname_colname_tabindid

where *tabindid* is a string concatenation of the table ID and index ID.

unique – constrains the values in the indicated column or columns so that no two rows can have the same non-null value. This constraint creates a unique index that can only be dropped if the constraint is dropped. You cannot use this option along with the **null** option described above.

primary key – constrains the values in the indicated column or columns so that no two rows can have the same value, and so that the value cannot be NULL. This constraint creates a unique index that can only be dropped if the constraint is dropped.

clustered | **nonclustered** – specifies that the index created by a **unique** or **primary key** constraint is a clustered or nonclustered index. **clustered** is the default (unless a clustered index already exists for the table) for primary key constraints, **nonclustered** is the default for unique constraints. There can be only one clustered index per table. See "**create index**" for more information.

fillfactor – specifies how full SQL Server will make each page when it is creating a new index on existing data. The **fillfactor** percentage is relevant only at the time the index is created. As the data changes, the pages are not maintained at any particular level of fullness.

The default for **fillfactor** is 0; this is used when you do not include **with fillfactor** in the **create index** statement (unless the value has been changed with **sp_configure**). When specifying a **fillfactor**, legal values are between 1 and 100.

A **fillfactor** of 0 creates clustered indexes with completely full pages and nonclustered indexes with completely full leaf pages. It leaves a comfortable amount of space within the index B-tree in both the clustered and nonclustered cases. There is seldom a reason to change the **fillfactor**.

If the **fillfactor** is set to 100, SQL Server creates both clustered and nonclustered indexes with each page 100% full. A **fillfactor** of 100 only makes sense for read-only tables—tables to which no additional data will ever be added.

fillfactor values smaller than 100 (except 0, which is a special case) cause SQL Server to create new indexes with pages that are not completely full. A **fillfactor** of 10 might be a reasonable choice if you are creating an index on a table that will eventually hold a great deal more data, but small **fillfactor** values cause each index (or index and data) to take more storage space.

◆ ***WARNING!***

Creating a clustered index with a fillfactor affects the amount of storage space your data occupies, since SQL Server redistributes the data as it creates the clustered index.

on *segment_name* – specifies that the index is to be created on the named segment. Before the **on** *segment_name* option can be used, the device must be initialized with **disk init**, and the segment must be added to the database with the **sp_addsegment** system procedure. See your System Administrator or use **sp_helpsegment** for a list of the segment names available in your database.

If you specify **clustered** and use the **on** *segment_name* option, the entire table migrates to the segment you specify, since the leaf level of the index contains the actual data pages.

references – specifies a column list for a referential integrity constraint. You can specify only one column value for a column-constraint. By including this constraint with a table that references another table, any data inserted into the "referencing" table must already exist in the "referenced" table.

To use this constraint, you must have **references** permission on the referenced table. The specified columns in the referenced table must be constrained by a unique index (created by either a **unique** constraint or a **create index** statement). If no columns are specified, there must be a **primary key** constraint on the appropriate columns in the referenced table. Also, the datatypes of the referencing table columns must exactly match the datatype of the referenced table columns.

foreign key – specifies that the listed column(s) are foreign keys in this table whose matching primary keys are the columns listed in the **references** clause.

ref_table – is the name of the table that contains the referenced columns. You can reference tables in another database.

ref_column – is the name of the column or columns in the referenced table.

check – specifies a *search_condition* constraint which SQL Server enforces for all the rows in the table.

search_condition – is a boolean expression that defines the **check** constraint on the column values. These constraints can include:

- a list of constant expressions introduced with **in**.
- a set of conditions, which may contain wildcard characters, introduced with **like**.

An expression can include arithmetic operations and Transact-SQL functions. The *search_condition* cannot contain subqueries, aggregate functions, parameters, or host variables.

next_column – indicates that you can include additional column definitions (separated by commas) using the same syntax described for a column definition.

drop – specifies the name of the constraint you want to drop from the table.

replace – specifies the column whose default value you want to change with the new value specified by a following **default** clause.

Examples

1.
```
alter table publishers
add manager_name varchar(40) null
```

   Adds a column to a table. For each existing row in the table, SQL Server assigns a NULL column value.

2.
```
alter table sales_daily
add ord_num numeric(5,0) identity
```

   Adds an IDENTITY column to a table. For each existing row in the table, SQL Server assigns a unique, sequential column value. Note that the IDENTITY column has type *numeric*, and scale zero. The precision determines the maximum value (10^5 - 1, or 99,999) that can be inserted into the column.

3.
```
alter table authors
add constraint au_identification
primary key (au_lname, au_fname, address)
```

   Adds a primary key constraint to the *authors* table. If there is an existing primary key or unique constraint on the table, the existing constraint must be dropped first. See Example 4.

4.
```
alter table titles
drop constraint au_identification
```

   Drops the *au_identification* constraint.

5. **alter table authors**
   replace phone default null

   Changes the default constraint on the *phone* column in the *authors* tables to insert the value NULL if the user does not enter a value.

Comments

Restrictions

- You cannot add a column of datatype *bit* to an existing table.
- The number of columns in a table cannot exceed 250. The maximum number of bytes per row is 1962.
- You cannot delete columns from a table. Use **select into** to create a new table with a different column structure or sort order.
- A column added with **alter table** is not visible to stored procedures that **select** * from that table. To reference the new column, drop the procedures and recreate them.

Getting Information About Tables

- For information on a table and its columns, use **sp_help**.
- To rename a table, execute the system procedure **sp_rename** (do not rename the system tables).
- For information about integrity constraints (**unique**, **primary key**, **references**, and **check**) or the **default** clause, see **create table** in this chapter.

Using Cross-Database Referential Integrity Constraints

- When you create a cross-database constraint, SQL Server stores the following information in the *sysreferences* system table of each database:

Information Stored in *sysreferences*	Columns with Information about the	
	Referenced Table	Referencing Table
Key Column IDs	*refkey1* through *refkey16*	*fokey1* through *fokey16*
Table ID	*reftabid*	*tableid*
Database Name	*pmrydbname*	*frgndbname*

Table 1-2: Information stored about referential integrity constraints

- You can drop the referencing table or its database without problems. SQL Server automatically removes the foreign key information from the referenced database.
- Because the referencing table depends on information from the referenced table, SQL Server does not allow you to:
  - Drop the referenced table,
  - Drop the external database that contains it, or
  - Rename either database with **sp_renamedb**

  You must first remove the cross-database constraint with **alter table.**
- Each time you add or remove a cross-database constraint, or drop a table that contains a cross-database constraint, dump **both** of the affected databases.

◆ ***WARNING!***

Loading earlier dumps of these databases could cause database corruption.

- The *sysreferences* system table stores the **name**—not the ID number—of the external database. SQL Server cannot guarantee referential integrity if you use **load database** to change the database name or to load it onto a different server.

◆ ***WARNING!***

Before dumping a database in order to load it with a different name or move it to another SQL Server, use alter table to drop all external referential integrity constraints.

Adding IDENTITY Columns

- When adding an IDENTITY column to a table, make sure the column precision is large enough to accommodate the number of existing rows. If the number of rows exceeds $10^{precision} - 1$, SQL Server generates an error when adding the column.
- When you add an IDENTITY column to a table, SQL Server:
  - Locks the table until all the IDENTITY column values have been generated. If a table contains a large number of rows, this process may be time consuming.
  - Assigns each existing row a unique, sequential IDENTITY column value, beginning with the value 1.
  - Logs each insert operation into the table. Use **dump transaction** to clear the database's transaction log before adding an IDENTITY column to a table with a large number of rows.
- Each time you insert a row into the table, SQL Server generates an IDENTITY column value one higher than the last. This value takes precedence over any defaults declared for the column in the **alter table** statement or bound to it with **sp_bindefault**.

Standards and Compliance

Standard	Compliance Level	Comments
SQL89	Transact-SQL extension	See "Datatypes" for datatype compliance information.
SQL92	Transact-SQL extension	

Permissions

alter table permission defaults to the table owner; it cannot be transferred except to the Database Owner, who can impersonate the table owner by running the **setuser** command. System Administrators can also alter users' tables.

See Also

Commands	**create table, drop table**
Topics	IDENTITY Columns
System procedures	**sp_help, sp_rename**

begin...end

Function

Encloses a series of SQL statements so that control-of-flow language, such as **if ... else**, can affect the performance of the whole group.

Syntax

```
begin
   statement block
end
```

Keywords and Options

statement block – a series of statements enclosed by **begin** and **end**.

Examples

1.
```
if (select avg(price) from titles) < $15
begin
  update titles
  set price = price * $2
  select title, price
  from titles
  where price > $28
end
```

   Without **begin** and **end**, the **if** condition would cause execution of only one SQL statement.

2.
```
create trigger deltitle
on titles
for delete
as
if
  (select count(*) from deleted, salesdetail
  where salesdetail.title_id = deleted.title_id) > 0
  begin
    rollback transaction
    print "You can't delete a title with sales."
  end
else
  print "Deletion successful--no sales for this
     title."
```

Comments

- **begin...end** blocks can nest within other **begin...end** blocks.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

See Also

Topics	Control-of-Flow Language

begin transaction

Function

Marks the starting point of a user-defined transaction.

Syntax

```
begin {transaction | tran } [transaction_name]
```

Keywords and Options

transaction_name – is the name assigned to this transaction. It must conform to the rules for identifiers. Use transaction names only on the outermost pair of nested **begin transaction** / **commit** or **begin transaction** / **rollback** statements.

Comments

- See "Transactions" in this chapter for full information on using transaction statements.
- Define a transaction by enclosing SQL statements and / or system procedures within the phrases **begin transaction** and **commit**. If you set chained transaction mode, SQL Server implicitly invokes a **begin transaction** before the following statements: **delete**, **insert**, **open**, **fetch**, **select**, and **update**. You must still explicitly close the transaction with a **commit**.
- To cancel all or part of a transaction, use the **rollback** command. The **rollback** command must appear within a transaction: you cannot roll back a transaction after it is committed.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

Permissions

begin transaction permission defaults to all users. No permission is required to use it.

See Also

Commands	**commit, rollback, save transaction**
Topics	Transactions

break

Function

Causes an exit from a **while** loop. **break** is often activated by an **if** test.

Syntax

```
while logical_expression
      statement
   break
      statement
   continue
```

Keywords and Options

logical_expression – is an expression (a column name, constant, any combination of column names and constants connected by arithmetic or bitwise operators, or a subquery) that returns TRUE, FALSE or NULL. If the logical expression contains a **select** statement, it must be enclosed in parentheses.

Examples

```
while (select avg(price) from titles) < $30
 begin
  update titles
  set price = price * 2
  select max(price) from titles
  if (select max(price) from titles) > $50
     break
  else
     continue
 end
 begin
  print "Too much for the market to bear"
 end
```

If the average price is less than $30, double the prices. Then select the maximum price. If it is less than or equal to $50, restart the **while** loop and double the prices again. If the maximum price is more than $50, exit the **while** loop and print a message.

Comments

- **break** causes an exit from a **while** loop. Statements that appear after the keyword **end**, which marks the end of the loop, are then executed.

- If two or more **while** loops are nested, the inner **break** exits to the next outermost loop. First all the statements after the end of the inner loop run, and then the next outermost loop restarts.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

See Also

Commands	**continue, while**
Topics	Control-of-Flow language, Expressions

checkpoint

Function

Writes all "dirty" pages (pages that have been updated since they were last written) to the database device.

Syntax

```
checkpoint
```

Examples

```
checkpoint
```

All "dirty" pages in the current database have been written to the database device, regardless of the system checkpoint schedule.

Comments

Automatic Checkpoints

- Checkpoints caused by the **checkpoint** command supplement automatic checkpoints, which occur at intervals calculated by SQL Server on the basis of the configurable value for maximum acceptable recovery time.
- The **checkpoint** shortens the automatic recovery process by identifying a point at which all completed transactions are guaranteed to have been written to the database device. A typical **checkpoint** takes about one second, though this figure varies depending on the amount of activity on SQL Server.
- The automatic **checkpoint** interval is calculated by SQL Server on the basis of system activity and the recovery interval value in the system table *syscurconfigs*. The recovery interval determines **checkpoint** frequency by specifying the amount of time it should take the system to recover. Reset this value by executing the system procedure **sp_configure** and the **reconfigure** command. For more information, see the *System Administration Guide*.

Manual Checkpoints

- Use **checkpoint** only as a precautionary measure in special circumstances. For example, you are instructed to issue the **checkpoint** command after resetting **select into/bulk copy.**

- Use **checkpoint** each time you change a database option with the system procedure **sp_dboption**. If you use **sp_dboption** inside a user-defined transaction, and then roll back that transaction, you must issue another **checkpoint** command in order to make the rollback take effect on the option change.

  For example:

```
begin tran
use master
go

sp_dboption "pubs2", "single", "true"
go
use pubs2
go
checkpoint
go

rollback tran
go
```

  If the following checkpoint is not issued, the *pubs*2 database remains single-user.

```
checkpoint
go
```

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

Permissions

checkpoint permission defaults to the Database Owner. It cannot be transferred.

See Also

Commands	reconfigure
System procedures	sp_dboption

close

Function

Deactivates a cursor.

Syntax

```
close cursor_name
```

Parameters

cursor_name – is the name of the cursor to close.

Examples

```
close authors_crsr
```

Closes the cursor named *authors_crsr*.

Comments

- SQL Server returns an error message if the cursor is already closed or does not exist.
- The **close** command essentially removes the cursor's result set. The cursor position within the result set is undefined for a closed cursor.

Standards and Compliance

Standard	Compliance Level
SQL89	Compliant
SQL92	Entry level compliant

See Also

Commands	deallocate cursor, declare cursor, fetch, open
Topics	Cursors

commit

Function

Marks the ending point of a user-defined transaction.

Syntax

```
commit [transaction | tran | work] [transaction_name]
```

Keywords and Options

transaction_name – is the name assigned to the transaction. It must conform to the rules for identifiers. Use transaction names only on the outermost pair of nested **begin transaction** / **commit** or **begin transaction** / **rollback** statements.

Comments

- See "Transactions" in Chapter 3 for full information on using transaction statements.
- Define a transaction by enclosing SQL statements and / or system procedures with the phrases **begin transaction** and **commit.** If you set the chained transaction mode, SQL Server implicitly invokes a **begin transaction** before the following statements: **delete**, **insert**, **open**, **fetch**, **select**, and **update**. You must still explicitly enclose the transaction with a **commit.**
- To cancel all or part of an entire transaction, use the **rollback** command. The **rollback** command must appear within a transaction; you cannot roll back a transaction after the **commit** has been entered.
- If no transaction is currently active, the **commit** or **rollback** statement has no effect on SQL Server.

Standards and Compliance

Standard	Compliance Level	Comments
SQL89	Compliant	The **commit transaction** and **commit tran** forms of the statement are Transact-SQL extensions.
SQL92	Entry level compliant	

Permissions

commit permission defaults to all users.

See Also

Commands	**begin transaction, rollback, save transaction**
Topics	Transactions

compute Clause

Function

Generates summary values that appear as additional rows in the query results. This allows you to see the detail and summary rows in one set of results. You can calculate summary values for subgroups, and you can calculate more than one aggregate for the same group.

Syntax

Start of **select** *statement*

```
compute row_aggregate (column_name)
    [, row_aggregate(column_name)]...
[by column_name [, column_name]...]
```

Keywords and Options

row_aggregate – is one of the following:

Function	Meaning
sum	Total of values in the (numeric) column
avg	Average of values in the (numeric) column
min	Lowest value in the column
max	Highest value in the column
count	Number of values in the column

Table 1-3: Row aggregate functions used with the compute clause

column_name – is the name of a column. It must be enclosed in parentheses. Only numeric columns can be used with **sum** and **avg**.

One **compute** clause can apply several aggregate functions to the same set of grouping columns (see examples 2 and 3). To create more than one group, use more than one **compute** clause (see example 5).

by – indicates that row aggregate values are to be calculated for subgroups. Whenever the value of the **by** item changes, row aggregate values are generated. If you use **by**, you must use **order by**.

Listing more than one item after **by** breaks a group into subgroups and applies a function at each level of grouping.

Examples

1.
```
select type, price
from titles
where price > $12
and type like "%cook"
order by type, price
compute sum(price) by type
```

```
type       price
---------  ------------
mod_cook          19.99
           sum
           ------------
                  19.99
type       price
---------  ------------
trad_cook         14.99
trad_cook         20.95
           sum
           ------------
                  35.94
(5 rows affected)
```

Calculates the sum of the prices of each type of cooking book that costs more than $12.

2.
```
select type, price, advance
from titles
where price > $12
and type like "%cook"
order by type, price
compute sum(price), sum(advance) by type
```

```
type       price      advance
---------  ---------  ------------
mod_cook       19.99          0.00
          sum         sum
           ---------  ------------
               19.99          0.00
type       price      advance
---------  ---------  ------------
trad_cook      14.99      8,000.00
trad_cook      20.95      7,000.00
           sum        sum
           ---------  ------------
               35.94     15,000.00
(5 rows affected)
```

Calculates the sum of the prices and advances for each type of cooking book costing over $12.

```
3. select type, price, advance
   from titles
   where price > $12
   and type like "%cook"
   order by type, price
   compute sum(price), max(advance) by type

type      price     advance
--------- --------- -------------
mod_cook      19.99         0.00
          sum
          ---------
              19.99
                    max
                    -------------
                             0.00

type      price     advance
--------- --------- -------------
trad_cook     14.99      8,000.00
trad_cook     20.95      7,000.00
          sum
          ---------
              35.94
                     max
                     -------------
                          8,000.00
(5 rows affected)
```

Calculates the sum of the prices and maximum advance of each type of cooking book costing over $12.

```
4. select type, pub_id, price
   from titles
   where price > $10
   and type = "psychology"
   order by type, pub_id, price
   compute  sum(price) by type, pub_id

type         pub_id    price
------------ --------- -----------
psychology   0736          10.95
psychology   0736          19.99
                          sum
                          ---------
                             30.94
```

```
type         pub_id    price
------------ --------- ---------
psychology   0877          21.59
                        sum
                        ---------
                            21.59
(5 rows affected)
```

Breaks on *type* and *pub_id* and calculates the sum of the prices of psychology books by type-publisher ID combination.

```
5. select type, pub_id, price
   from titles
   where price > $10
   and type = "psychology"
   order by type, pub_id, price
   compute  sum(price) by type, pub_id
   compute  sum(price) by type
```

```
type         pub_id    price
------------ --------- ---------
psychology   0736          10.95
psychology   0736          19.99
                        sum
                        ---------
                            30.94

type         pub_id    price
------------ --------- ---------
psychology   0877          21.59
                        sum
                        ---------
                            21.59
                        sum
                        ---------
                            52.53
(6 rows affected)
```

Calculates the grand total of the prices of psychology books that cost more than $10 in addition to sums by *type* and *pub_id*.

```
6. select type, price, advance
   from titles
   where price > $10
   and type like "%cook"
   compute sum(price), sum(advance)
```

```
type       price        advance
---------  -----------  --------------
mod_cook         19.99            0.00
trad_cook        20.95        8,000.00
trad_cook        11.95        4,000.00
trad_cook        14.99        7,000.00
            sum          sum
           -----------  --------------
                 67.88       19,000.00
(5 rows affected)
```

Calculates the grand totals of the prices and advances of cooking books that cost more than $10.

7.
```
select type, price, price*2
from titles
where type like "%cook"
compute sum(price), sum(price*2)
```

```
type         price
------------ -------------  ------------
mod_cook              19.99         39.98
mod_cook               2.99          5.98
trad_cook             20.95         41.90
trad_cook             11.95         23.90
trad_cook             14.99         29.98
             sum              sum
             =============    ============
                     70.87         141.74
```

Calculates the sum of the price of cooking books, and the sum of the price used in an expression.

Comments

Restrictions

- You cannot use a **compute** clause in a cursor declaration.
- Summary values can be computed for expressions as well as for columns. Any expression or column that appears in the **compute** clause must appear in the **select** list.
- Aliases for column names are not allowed as arguments to the row aggregate in a **compute** clause, although they can be used in the **select** list, the **order by** clause, and the **by** clause of **compute**.
- You cannot use **select into** in the same statement as a **compute** clause, because statements that include **compute** do not generate normal tables.

- If you use **compute by**, you must also use an **order by** clause. The columns listed after **compute by** must be identical to or a subset of those listed after **order by**, and must be in the same left-to-right order, start with the same expression, and not skip any expressions. For example, if the **order by** clause is:

  ```
  order by a, b, c
  ```

  the **compute by** clause can be any (or all) of these:

  ```
  compute by a, b, c
  compute by a, b
  compute by a
  ```

- The **compute** keyword can be used without **by** to generate grand totals, grand counts, and so on. **order by** is optional if you use the **compute** keyword without **by**. See Example 6.

- In a **select** statement with a **compute** clause, the order of columns in the select list overrides the order of the aggregates in the **compute** clause. DB-Library programmers must be aware of this in order to put the aggregate results in the right place. See "Row Aggregates" for an example.

Results Appear as a New Row

- The aggregate functions ordinarily produce a single value for all the selected rows in the table or for each group, and these summary values are shown as new columns. For example:

  ```
  select type, sum(price), sum(advance)
  from titles
  where type like "%cook"
  group by type
  ```

  ```
  type
  ------------  ---------  ----------
  mod_cook           22.98  15,000.00
  trad_cook          47.89  19,000.00

  (2 rows affected)
  ```

- The **compute** clause makes it possible to retrieve detail and summary rows with one command. For example:

  ```
  select type, price, advance
  from titles
  where type like "%cook"
  order by type
  compute sum(price), sum(advance) by type
  ```

```
type       price      advance
---------  ---------  --------------
mod_cook        2.99       15,000.00
mod_cook       19.99            0.00
           sum        sum
           ---------  --------------
               22.98       15,000.00

type       price      advance
---------  ---------  --------------
trad_cook      11.95        4,000.00
trad_cook      14.99        8,000.00
trad_cook      20.95        7,000.00
           sum        sum
           ---------  --------------
               47.89       19,000.00

(7 rows affected)
```

Case Sensitivity

- If your server has a case-insensitive sort order installed, **compute** ignores the case of the data in the columns you specify. For example, given this data:

  ```
  select * from groupdemo
  lname      amount
  ---------- ------------------
  Smith                  10.00
  smith                   5.00
  SMITH                   7.00
  Levi                    9.00
  Lévi                   20.00
  ```

  Using **compute by** on *lname* produces these results:

  ```
  select lname, amount from groupdemo
  order by lname
  compute sum(amount) by lname
  ```

```
lname      amount
---------- -------------------------
Levi                          9.00
           sum
           -------------------------
                              9.00

lname      amount
---------- -------------------------
Lévi                         20.00
           sum
           -------------------------
                             20.00

lname      amount
---------- -------------------------
smith                         5.00
SMITH                         7.00
Smith                        10.00
           sum
           -------------------------
                             22.00
```

The same query on a case- and accent insensitive server produces these results:

```
lname      amount
---------- -------------------------
Levi                          9.00
Lévi                         20.00
           sum
           -------------------------
                             29.00

lname      amount
---------- -------------------------
smith                         5.00
SMITH                         7.00
Smith                        10.00
           sum
           -------------------------
                             22.00
```

Standards and Compliance

Standard	Compliance level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

See Also

Commands	**group by** and **having** clauses, **select**
Functions	Aggregate Functions, Row Aggregate Functions

continue

Function

Causes the **while** loop to restart. **continue** is often activated by an **if** test.

Syntax

```
while boolean_expression
      statement
  break
      statement
  continue
```

Examples

```
while (select avg(price) from titles) < $30
begin
  update titles
  set price = price * 2
  select max(price) from titles

  if (select max(price) from titles) > $50
      break
  else
      continue
end

begin
print "Too much for the market to bear"
end
```

If the average price is less than $30, double the prices. Then select the maximum price. If it is less than or equal to $50, restart the **while** loop and double the prices again. If the maximum price is more than $50, exit the **while** loop and print a message.

Comments

- **continue** causes the **while** loop to restart, skipping any statements after **continue**.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

See Also

Commands	**break, while**
Topics	Control-of-Flow Language, Expressions

create database

Function

Creates a new database. Use **create database** from the *master* database.

Syntax

```
create database database_name
   [on {default | database_device} [= size]
       [, database_device [= size]]...]
   [log on database_device [= size]
       [, database_device [= size]]...]
   [with override]
   [for load]
```

Keywords and Options

database_name – is the name of the new database. It must conform to the rules for identifiers.

on – is a keyword indicating that you want to specify a location and (optionally) a size for the database.

default – indicates that **create database** can put the new database on any default database device(s) (as shown in *sysdevices.status*). To specify a size for the database without specifying a location, use this command:

```
on default = size
```

To change a database device's status to default, use the system procedure **sp_diskdefault**.

database_device – is the logical name of the device on which you want to locate the database. A database can occupy different amounts of space on each of several database devices.

size – is the amount of space allocated to the database in megabytes. SQL Server-supplied default size is 2 MB. System Administrators can increase the default size by using **sp_configure**, issuing the **reconfigure** command, and restarting SQL Server. The default size must be at least as large as the *model* database. If you increase the size of the *model* database, the default size must also be increased. The legal values for database size range from 2 to 2048 MB.

If SQL Server cannot give you as much space as you want where you have requested it, it comes as close as possible on a per-

device basis, and prints a message telling how much space was allocated where.

log on – specifies the logical name of the device that will be used to store the database logs. You can specify more than one device in the **log on** clause.

with override– allows you to specify the same device name in the **on** and **log on** clauses, even if this creates a device fragment with data and a device fragment with log on the same device. You can still use **dump transaction** on the log segment, but if the device fails, you cannot use the **with no_truncate** option to recover changes made since your last dump. If you attempt to mix log and data on the same device without this clause, the **create database** command fails. If you mix log and data, and use **with override**, you are warned, but the command succeeds.

for load – invokes a streamlined version of **create database** that can only be used for loading a database dump. Use this option for recovery from media failure or for moving a database from one machine to another.

Examples

```
1. create database pubs

2. create database pubs
   on default = 4

3. create database pubs
   on datadev = 3, moredatadev = 2

4. create database pubs
   on datadev = 3
   log on logdev = 1
```

Comments

Restrictions

- SQL Server can manage up to 32,767 databases.
- Only one database can be created at a time. If two database creation requests collide, one user will get this message:

  ```
  model database in use: cannot create new database
  ```

- The maximum number of device fragments for any database is 128. Each time you allocate space on a database device with **create**

database or **alter database**, that allocation represents a device fragment, and the allocation is entered as a row in *sysusages*.

- The maximum number of named segments for any database is 32. Segments are named subsets of database devices available to a particular SQL Server. For more information on segments, see the *System Administration Guide*.
- If you do not specify a location and size for a database, the default location is any default database device(s) indicated in *master..sysdevices*. The default size is the larger of the size of the *model* database or the **database size** parameter in *sysconfigures*.

New Databases Are Created from model

- SQL Server creates a new database by copying the *model* database.
- You can customize *model* by adding tables, stored procedures, user-defined datatypes, and other objects and by changing database option settings. New databases inherit these objects and settings from *model*.
- To guarantee recoverability, the **create database** command must clear every page that was not initialized when the *model* database was copied. This may take several minutes, depending on the size of the database and the speed of your system.

  If you are creating a database in order to load a database dump into it, you can use the **for load** option to skip the page clearing step. This makes database creation considerably faster.

Ensuring Database Recoverability

- Back up the *master* database each time you create a new database. This makes recovery easier and safer in case *master* is damaged.

➤ ***Note***

If you create a database and fail to back up *master*, you may be able to recover the changes with **disk refit**.

- The **with override** clause allows you to mix log and data segments on a single device. However, for full recoverability, the device or devices specified in **log on** should be different than the physical device that stores the data. In the event of a hard disk crash, the database can be recovered from database dumps and the transaction logs. A small database can be created on a single

device, and the transaction log stored together with the rest of the database, but you **must** rely on the **dump database** command for backups.

- The size of the device required for the transaction log varies according to the amount of update activity and the frequency of transaction log dumps. As a rule of thumb, allocate to the log device 10% to 25% of the space you allocate to the database itself. It is best to start small, since space allocated to a transaction log device cannot be reclaimed and cannot be used for storage of data.
- If you create a database using the **for load** option, you can run only the following commands in the new database before loading a database dump:
  - **alter database for load**
  - **drop database**
  - **load database**

  Use **alter database for load** to create the new database in the image of the database from which the database dump to be loaded was made. See the *System Administration Guide* for a discussion of duplicating space allocation when loading a dump into a new database.

  After you load the database dump into the new database, there are no restrictions on the commands you can use.

Getting Information About Databases

- To get a report on a database, execute the system procedure **sp_helpdb**.
- For a report on the space used in a database, use **sp_spaceused**.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

Permissions

create database permission defaults to System Administrators, who can transfer it to users who are listed in the *sysusers* table of the

current database. However, **create database** permission is often centralized in order to maintain control over database storage allocation. See the *System Administration Guide* for details.

create database permission is not included in the **grant all** command.

See Also

Commands	**alter database, drop database, dump database, load database**
System procedures	**sp_changedbowner, sp_diskdefault, sp_helpdb, sp_logdevice, sp_renamedb, sp_spaceused**

create default

Function

Specifies a value to insert in a column (or in all columns of a user-defined datatype) if no value is explicitly supplied at insert time.

Syntax

```
create default [owner.]default_name
   as constant_expression
```

Keywords and Options

default_name – is the name of the default. It must conform to the rules for identifiers.

constant_expression – is an expression that does not include the names of any columns or other database objects. You can include built-in functions that do not reference database objects. Enclose character and date constants in quotes and precede binary constants with *0x*.

Examples

1. `create default phonedflt as "UNKNOWN"`

   Defines a default value. Now you need to bind it to the appropriate column or user-defined datatype with the **sp_bindefault** system stored procedure.

2. `sp_bindefault phonedflt, "authors.phone"`

   The default takes effect only if there is no entry in the *phone* column of the *authors* table. "No entry" is different than entering a null value. To get the default using **isql**, you must issue an **insert** command with a column list that does not include the column that has the default.

3. `create default todays_date as getdate()`

   Creates a default value, *todays_date*, that inserts the current date into columns to which it is bound.

Comments

Restrictions

- You can only create a default in the current database.

- Bind a default to a column or user-defined datatype—but not a SQL Server-supplied datatype— with **sp_bindefault.**
- **create default** statements cannot be combined with other statements in a single batch.
- You must **drop** a default before you **create** a new one of the same name, and you must unbind a default (with the system procedure **sp_unbindefault**) before you **drop** it.
- You can bind a new default to a column or datatype without unbinding the old one: the new default overrides and automatically unbinds the old one.
- You can bind one default to a user-defined datatype, and another default to a column of that datatype. The default most recently bound takes precedence.

Datatype Compatibility

- SQL Server generates an error message when it tries to insert a default value that is not compatible with the column's datatype. For example, if you bind the default "N/A" to an *integer* column, any **insert** that does not specify the column value will fail.
- If a default value is too long for a character column, SQL Server either truncates the string or generates an exception, depending on the setting of the **string_rtruncation** option. For more information, see the **set** command.

Getting Information About Defaults

- Default definitions are stored in *syscomments*.
- After a default is bound to a particular column its object ID is stored in *syscolumns*. After a default is bound to a user-defined datatype, its object ID is stored in *systypes*.
- To display definitions and binding information, execute the system procedure **sp_help** with the default name as the parameter.
- To rename a default, use **sp_rename**.
- For a report on the text of a default, use **sp_helptext**.

Defaults and Rules

- If a column has both a default and a rule associated with it, the default value must not violate the rule. A default that conflicts

with a rule will never be inserted. SQL Server will generate an error message each time it attempts to insert such a default.

Defaults and Nulls

- If you specify NOT NULL when you create a column and do not create a default for it, an error message will display whenever a user fails to make an entry in that column.
- The following table illustrates the relationship between the existence of a default and the definition of a column as NULL or NOT NULL. The entries in the table show the result:

	No Entry, No Default	No Entry, Default	Enter NULL, No Default	Enter NULL, Default
NULL	null	default	null	null
NOT NULL	error	default	error	error

Table 1-4: Relationship between nulls and column defaults

Specifying a Default Value in create table

- You can also define column defaults using the **default** clause of the **create table** statement as an alternative to using **create default.** However, these column defaults are specific for that table; you cannot bind them to other tables. See "**create table**" and "**alter table**" for information about integrity constraints.

Standards and Compliance

Standard	Compliance Level	Comments
SQL89	Transact-SQL extension	Use the **default** clause of the **create table** statement to create defaults that are SQL 89 and SQL92 compliant.
SQL92	Transact-SQL extension	

Permissions

create default permission defaults to the Database Owner, who can transfer it to other users.

See Also

Commands	**alter table, create rule, create table, drop default, drop rule**
Topics	Batch Queries
System procedures	**sp_bindefault, sp_help, sp_helptext, sp_rename, sp_unbindefault**

create index

Function

Creates an index on one or more columns in a table.

Syntax

```
create [unique] [clustered | nonclustered]
       index index_name
   on [[database.]owner.]table_name (column_name
       [, column_name]...)
   [with {fillfactor = x, ignore_dup_key, sorted_data,
       [ignore_dup_row | allow_dup_row]}]
   [on segment_name]
```

Keywords and Options

unique – prohibits duplicate index (also called key) values. The system checks for duplicate key values when the index is created (if data already exists), and checks each time data is added with an **insert** or **update**. If there is a duplicate key value or if more than one row contains a null value, the command is aborted and an error message giving the duplicate is printed.

◆ ***WARNING!***

SQL Server does not detect duplicate rows if a table contains any non-null *text* or *image* columns.

update or **insert** commands that generate duplicate key values are canceled, unless the index was created with **ignore_dup_row** or **ignore_dup_key.**

Composite indexes (indexes in which the key value is composed of more than one column) can also be unique.

The default is non-unique.

clustered –means the physical order of rows on this database device is the same as the indexed order of the rows. The bottom or leaf level of the clustered index contains the actual data pages. A clustered index almost always retrieves data faster than a nonclustered index. Only one clustered index is permitted per table.

The clustered index is often created on the table's primary key—the column or columns that uniquely identify the row. The primary key can be recorded in the database (for use by front end programs and the system procedure **sp_depends**) with the system procedure **sp_primarykey.** If **clustered** is not specified, **nonclustered** is assumed.

nonclustered – means that there is a level of indirection between the index structure and the data itself. You can have up to 249 nonclustered indexes per table.

index_name – is the name of the index. Index names must be unique within a table, but need not be unique within a database.

table_name – is the name of the table in which the indexed column or columns are located.

column_name – is the column or columns to which the index applies. Composite indexes are based on the combined values of up to 16 columns. The sum of the maximum lengths of all the columns used in a composite index cannot exceed 256 bytes. List the columns to be included in the composite index (in sort-priority order) inside the parentheses after *table_name*.

fillfactor – specifies how full SQL Server will make each page when it is creating a new index on existing data. The **fillfactor** percentage is relevant only at the time the index is created. As the data changes, the pages are not maintained at any particular level of fullness.

The default for **fillfactor** is 0; this is used when you do not include **with fillfactor** in the **create index** statement (unless the value has been changed with **sp_configure**). When specifying a **fillfactor**, legal values are between 1 and 100.

A **fillfactor** of 0 creates clustered indexes with completely full pages and nonclustered indexes with completely full leaf pages. It leaves a comfortable amount of space within the index B-tree in both the clustered and nonclustered cases. There is seldom a reason to change the **fillfactor.**

If the **fillfactor** is set to 100, SQL Server creates both clustered and nonclustered indexes with each page 100% full. A **fillfactor** of 100 only makes sense for read-only tables—tables to which no additional data will ever be added.

fillfactor values smaller than 100 (except 0, which is a special case) cause SQL Server to create new indexes with pages that are not completely full. A **fillfactor** of 10 might be a reasonable choice if

you are creating an index on a table that will eventually hold a great deal more data, but small **fillfactor** values cause each index (or index and data) to take more storage space.

◆ ***WARNING!***

Creating a clustered index with a fillfactor affects the amount of storage space your data occupies, since SQL Server redistributes the data as it creates the clustered index.

ignore_dup_key – responds to a duplicate key entry into a table that has a unique index (clustered or nonclustered). An attempted **insert** of a duplicate key is ignored, and the **insert** is canceled with an informational message.

If **ignore_dup_key** is in effect, a transaction that contains duplicate keys will proceed to completion, and informational messages about the presence of duplicate keys will appear.

➤ ***Note***

When **ignore_dup_key** is in effect, an attempted update that creates a duplicate key causes that **update** to be canceled. After the cancellation, any transaction that may have been active at the time may continue as though the **update** had never taken place.

You cannot create a unique index on a column that includes duplicate values or more than one null value, whether or not **ignore_dup_key** is set. If you attempt to do so, SQL Server prints an error message that gives the first of the duplicate values. You must eliminate duplicates before you create a unique index on the column.

ignore_dup_row and **allow_dup_row** – are options for creating a non-unique, clustered index. These options are not relevant when creating a non-unique, nonclustered index. (Since a SQL Server nonclustered index attaches a unique row identification number internally, it never worries about duplicate rows—even for identical data values.)

ignore_dup_row and **allow_dup_row** are mutually exclusive.

Including the **allow_dup_row** option allows you to create a new, non-unique, clustered index on a table that includes duplicate rows. If a table has a nonunique clustered index that was created

without the **allow_dup_row** option, you cannot create new duplicate rows using the **insert** or **update** command.

If any index in the table is unique, the requirement for uniqueness takes precedence over the **allow_dup_row** option. You cannot create an index with **allow_dup_row** if a unique index exists on any column in the table.

The **ignore_dup_row** option is also used with a non-unique, clustered index. The **ignore_dup_row** option is used to eliminate duplicates from a batch of data. **ignore_dup_row** cancels any **insert** or **update** that would create a duplicate row, but does not roll back the entire transaction.

The **ignore_dup_row** option is not allowed if a unique index exists on any column in the table.

This table illustrates how **allow_dup_row** and **ignore_dup_row** affect attempts to create a non-unique clustered index on a table that includes duplicate rows, and to enter duplicate rows into a table.

	create index (table has duplicate rows)	**Insert duplicate rows**
Neither option set	Command fails.	Command fails.
allow_dup_row set	Command completes.	Command completes.
ignore_dup_row set	Index created but duplicate rows thrown out; error message.	All rows accepted except duplicates; error message. See warning.

Table 1-5: Duplicate row options

The following table shows how index options can be used:

Index Type	**Options**
clustered	**ignore_dup_row** | **allow_dup_row**
unique clustered	**ignore_dup_key**
nonclustered	none
unique nonclustered	**ignore_dup_key**, **ignore_dup_row**

Table 1-6: Index options

sorted_data – speeds creation of an index when the data in the table is already in sorted order (for example, when you have used **bcp** to copy data that has already been sorted into an empty table). The speed increase becomes significant on large tables and increases

to several times faster in tables larger than a gigabyte. This option can be used in conjunction with any other **create index** options with no effect on their operation.

If **sorted_data** is specified but data is not in sorted order, an error message displays and the command is aborted.

This option speeds indexing only for clustered indexes or unique nonclustered indexes. Creating a non-unique nonclustered index succeeds unless there are rows with duplicate keys. If there are rows with duplicate keys, an error message displays and the command is aborted.

on *segment_name* – specifies that the index is to be created on the named segment. Before the **on** *segment_name* option can be used, the device must be initialized with **disk init**, and the segment must be added to the database with the **sp_addsegment** system procedure. See your System Administrator or use **sp_helpsegment** for a list of the segment names available in your database.

If you specify **clustered** and use the **on** *segment_name* option, the entire table migrates to the segment you specify, since the leaf level of the index contains the actual data pages.

Examples

```
1. create index au_id_ind
   on authors (au_id)

2. create unique clustered index au_id_ind
   on authors(au_id)

3. create index ind1
   on titleauthor (au_id, title_id)

4. create nonclustered index zip_ind
   on authors(zip)
   with fillfactor = 25
```

Comments

Restrictions

- You cannot create an index on a column with a datatype of *bit*, *text*, or *image*.
- A table can have a maximum of 249 nonclustered indexes.
- A table can have a maximum of one clustered index.

- You can create an index on a temporary table. It disappears when the table disappears.
- You can create an index on a table in another database, as long as you are the owner of that table.
- You cannot create an index on a view.
- **create index** runs more slowly while a **dump database** is taking place.

Creating Indexes Efficiently

- Indexes speed data retrieval but can slow data update. Performance improvements can be realized by creating a table on one segment, and its nonclustered indexes on another segment, when these segments are on separate physical devices.
- By default, SQL Server reads and writes single data pages while creating indexes. A System Administrator can allocate buffers that allow **create index** to read and write 8 data pages (called an extent) at a time. This increases performance if you need to create indexes on large tables. See **sp_configure** in Volume 2 for more information.
- Create a clustered index before creating any nonclustered indexes, since nonclustered indexes are automatically rebuilt when a clustered index is created.
- Index all columns that are regularly used in joins.
- SQL Server runs **update statistics** each time an index is created on existing data. You should run **update statistics** periodically if you add data to the table that changes the distribution of keys in the index. The query optimizer uses the information created by **update statistics** to select the best plan for running queries on the table.

Space Requirements for Indexes

- Space is allocated to tables and indexes in increments of one extent, or 8 pages, at a time. Each time an extent is filled, another extent is allocated. (Use the system procedure **sp_spaceused** to display the amount of space allocated and used by an index.)
- A table "follows" its clustered index. When you create a table and then use the **on** *segment_name* extension to **create clustered index**, the table migrates to the segment where the index is created.
- To create a clustered index, SQL Server duplicates the existing data; the server deletes the original data when the index is complete. Before creating a clustered index, use **sp_spaceused** to

make sure that the database has at least 120% of the size of the table available as free space.

- The **sorted_data** option does **not** reduce the amount of space required to create an index.

Getting Information About Tables and Indexes

- Each index—including composite indexes— is represented by one row in *sysindexes*.
- For information on the order of the data retrieved through indexes, and the effects of a SQL Server's installed sort order, see the **order by** clause.
- For information about a table's indexes, execute the system procedure **sp_helpindex**.

Using Unique Constraints in Place of Indexes

- As an alternative to **create index**, you can implicitly create unique indexes by specifying a unique constraint with the **create table** or **alter table** statements. The unique constraint creates a clustered or non-clustered unique index on the columns of a table. These "implicit" indexes are named after the constraint and they follow the same rules for indexes created using **create index**.
- You cannot drop indexes supporting unique constraints using the **drop index** statement. Instead, they are dropped when the constraints are dropped through an **alter table** statement or when the table is dropped. See **create table** for more information about unique constraints.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

Permissions

create index permission defaults to the table owner, and is not transferable.

See Also

Commands	**alter table, create table, drop index, insert, order by clause, reconfigure, set, update, update statistics**
System procedures	**sp_addsegment, sp_helpindex, sp_helpsegment, sp_spaceused**

create procedure

Function

Creates a stored procedure that can take one or more user-supplied parameters.

Syntax

```
create procedure [owner.]procedure_name[;number]
   [[(]@parameter_name
       datatype [(length) | (precision [, scale])
       [= default][output]
   [, @parameter_name
       datatype [(length) | (precision [, scale])
       [= default][output]]...[)]]
   [with recompile]
   as SQL_statements
```

Keywords and Options

procedure_name – is the name of the procedure. It must conform to the rules for identifiers.

;number – is an optional integer used to group procedures of the same name so that they can be dropped together with a single **drop procedure** statement. Procedures used in the same application are often grouped this way. For example, if the procedures used with the application *orders* are named *orderproc;1*, *orderproc;2*, and so on, the statement:

```
drop proc orderproc
```

drops the entire group.

Once procedures have been grouped, individual procedures within the group cannot be dropped. For example, the statement:

```
drop procedure orderproc;2
```

is not allowed.

parameter_name – is the name of an argument to the procedure. The value of each parameter is supplied when the procedure is executed. (Parameter names are optional in **create procedure** statements—a procedure need not take any arguments.)

Parameter names must be preceded by the "@" symbol and conform to the rules for identifiers. A parameter name, including

the "@" symbol, can be a maximum of 30 characters. Parameters are local to the procedure: the same parameter names can be used in other procedures.

If the value of a parameter contains non-alphanumeric characters, it must be enclosed in quotes. This includes object names qualified by a database name or owner name, since they include a period. If the value of a character parameter begins with a numeric character, it also must be enclosed in quotes.

datatype [(*length)* | (*precision* [, *scale*])] – is the datatype of the parameter. See the "Datatypes" section in Chapter 3, ''Transact-SQL Topics'' for more information about datatypes. Stored procedure parameters cannot have a datatype of *text* or *image*, or a user-defined datatype whose underlying type is *text* or *image*.

The *char*, *varchar*, *nchar*, *nvarchar*, *binary*, and *varbinary* datatypes should include a *length* in parentheses. If you omit the length, SQL Server truncates the parameter value to one character.

The *float* datatype expects a binary *precision* in parentheses. If you omit the precision, SQL Server uses the default precision for your platform.

The *numeric* and *decimal* datatypes expect a *precision* and *scale*, in parentheses and separated by a comma. If you omit the precision and scale, SQL Server uses a default precision of 18 and scale of 0.

default – defines a default parameter value for the procedure. If a default is defined, a user can execute the procedure without giving a parameter. The default must be a constant. It can include the wildcard characters (%, _, [ ], and [^]) if the procedure uses the parameter name with the keyword **like**. (See example 2.)

The default can be NULL. The procedure definition can specify that some action be taken if the parameter value is NULL. (See example 3.)

output – indicates that the parameter is a return parameter. Its value can be returned to the **execute** command that called this procedure. Use return parameters to return information to the calling procedure. (See example 5.)

To return a parameter value through several level of nested procedures, each procedure must include the **output** option with the parameter name, including the **execute** command that calls the highest level procedure.

with recompile – means that SQL Server never saves a plan for this procedure; a new plan is created each time it is executed. Use this optional clause when you expect that the execution of the procedure won't be typical—that is, you need a new plan.

SQL_statements – specify the actions the procedure is to take. Any number and kind of SQL statements can be included, with the exception of **create view**, **create default**, **create rule**, **create procedure**, **create trigger**, or **use** statements.

create procedure SQL statements often include control-of-flow language, including one or more of the following: **declare**; **if-else** statement; **while**; **break**; **continue**; **begin...end**; **goto** label; **return**; **waitfor**; */*comment */*. They can also refer to parameters defined for the procedure.

The SQL statements can reference objects in another database, as long as they are properly qualified.

Examples

1. ```
   create procedure showind @tabname varchar(30)
   as
     select sysobjects.name, sysindexes.name, indid
     from sysindexes, sysobjects
     where sysobjects.name = @tabname
     and sysobjects.id = sysindexes.id
   ```

   Given a table name, the procedure *showind* displays its name and the names and identification numbers of any indexes on any of its columns.

   Here are the acceptable syntax forms for executing *showind:*

   ```
   execute showind titles
   ```

   ```
   execute showind @tabname = "titles"
   ```

   Or, if this is the first statement in a file or batch:

   ```
   showind titles
   ```

2. ```
   create procedure
   showsysind @table varchar(30) = "sys%"
   as
     select sysobjects.name, sysindexes.name, indid
     from sysindexes, sysobjects
     where sysobjects.name like @table
     and sysobjects.id = sysindexes.id
   ```

   This procedure displays information about the system tables if the user does not supply a parameter.

3.
```
create procedure
showindnew @table varchar(30) = null
as
  if @table is null
   print "Please give a table name"
  else
   select sysobjects.name, sysindexes.name, indid
   from sysindexes, sysobjects
   where sysobjects.name = @table
   and sysobjects.id = sysindexes.id
```

   This procedure specifies an action to be taken if the parameter is NULL (that is, if the user does not give a parameter).

4.
```
create procedure mathtutor @mult1 int, @mult2 int,
  @result int output
as
select @result = @mult1 * @mult2
```

   This stored procedure multiplies two integer parameters and returns the product in the **output** parameter *@result*. If the procedure is executed by passing it 3 integers, the **select** statement performs the multiplication and assigns the values, but does not print the return parameter:

```
mathtutor 5, 6, 32

(return status 0)
```

5.
```
declare @guess int
select @guess = 32
exec mathtutor 5, 6, @result = @guess output

(1 row affected)
(return status = 0)

Return parameters:

@result
-----------
         30
```

   In this example, the procedure and the **execute** statement both include the **output** option with a parameter name so the procedure can return a value to the caller. Note that the **output** parameter and any subsequent parameters in the **execute** statement, *@result*, **must** be passed as:

   @parameter = value

   - The value of the return parameter is always reported, whether its value has changed or not.

- *@result* does not need to be declared in the calling batch because it is the name of a parameter to be passed to *mathtutor*.
- Although the changed value of *@result* is returned to the caller in the variable assigned in the **execute** statement (in this case *@guess*), it is displayed under its own heading (*@result*).

```
6. declare @guess int
   declare @store int
   select @guess = 32
   select @store = @guess
   execute mathtutor 5, 6, @result = @guess output
   select Your_answer = @store, Right_answer = @guess
   if @guess = @store
       print "Right-o"
   else
       print "Wrong, wrong, wrong!"

   (1 row affected)
   (1 row affected)
   (return status = 0)

   Return parameters:

   @result
   -----------
            30

   Your_answer Right_answer
   ----------- ------------
            32           30

   (1 row affected)
   Wrong, wrong, wrong!
```

Return parameters can be used in additional SQL statements in the batch or calling procedure. This example shows how to use the value of *@guess* in conditional clauses after the **execute** statement, by storing it in another variable name, *@store*, during the procedure call. When return parameters are used in an **execute** statement that is part of a SQL batch, the return values are printed with a heading before subsequent statements in the batch are executed.

Comments

Restrictions

- The maximum number of parameters that a stored procedure can have is 255.
- The maximum number of local and global variables in a procedure is limited only by available memory.
- The maximum amount of text in a stored procedure is 16 MB.
- A **create procedure** statement cannot be combined with other statements in a single batch.
- You can only create a stored procedure in the current database, although the procedure can reference objects from other databases. Any objects referenced in a procedure must exist at the time you create the procedure. You can create an object within a procedure and then reference it, as long as the object is created before it is referenced.
- You cannot create a table and insert data into the table in the same batch, because when the **insert** statement is compiled the table does not yet exist.
- If you use "**select** *" in your **create procedure** statement, the procedure (even if you use the **with recompile** option to **execute**) does not pick up any new columns you may have added to the table. You must **drop** the procedure and recreate it.
- Within a stored procedure, you cannot create an object (including a temporary table), drop it, and then create a new object with the same name. SQL Server creates the objects defined in a stored procedure when the procedure is executed, not when it is compiled.

◆ ***WARNING!***

Certain changes to databases, such as dropping and recreating indexes, can cause object IDs to change. Stored procedures recompile automatically in this case, and can grow slightly in size. You should always leave some space for this increase in size.

Executing Stored Procedures

- Once a procedure is created, you can run it by issuing the **execute** command along with the procedure's name and any parameters.

If a procedure is the first statement in a batch, you can give its name without the keyword **execute**.

Creating New System Procedures

- System Administrators can write system procedures that can be executed from any database. These procedures must be created in the *sybsystemprocs* database and their names must begin with the characters **sp_**. For more information about creating system procedures, see ''Overview of System Administration'' in the *System Administration Guide*.

Object Identifiers

- To rename a procedure, use **sp_rename**.
- If a procedure references table names, column names, or view names that are not valid identifiers, you must **set quoted_identifier on** before the **create procedure** command and enclose each such name in double quotes. The **quoted_identifier** option does **not** need to be on when you execute the procedure.
- Column headings in **select** statements in procedures that will be called by APT-SQL, must conform to the rules for identifiers.
- You must drop and recreate the procedure if any of the objects it references have been renamed.
- Inside a stored procedure, object names used with the **alter table**, **create table**, **drop table**, **truncate table**, **create index**, **drop index**, **update statistics**, and **dbcc** commands must be qualified with the object owner's name if other users are to make use of the stored procedure. For example, user "mary," who owns table *marytab*, should qualify the name of her table inside a stored procedure (when it is used with these commands) if she wants other users to be able to execute it, like this:

  ```
  create procedure p1
  as
  create index marytab_ind
  on mary.marytab(col1)
  ```

  This is because the object names are resolved when the procedure is run. In the example, if user "john" tries to execute procedure *p1*, SQL Server looks for a table called *marytab* owned by the user "mary."

Object names used with other statements (for example, **select** or **insert**) inside a stored procedure need not be qualified, because the names are resolved when the procedure is compiled.

Permissions on Objects

- The owner of the stored procedure must own the tables and views used in the procedure. Not even the System Administrator can create a stored procedure to perform operations on another user's tables, if he or she has not been granted permission on those tables.
- A table or view owner can create procedures that perform operations on his or her objects. Users who execute that procedure need not have permissions to those objects directly. For example, a table or view owner can create a procedure that contains **select**, **insert** or **update** statements that other users do not have permission to run directly. This provides a security mechanism: a user can run a procedure that displays certain columns from a table, but not all of the data.

Temporary Tables and Procedures

- You can create a procedure to reference a temporary table if the temporary table is created in the current session. A temporary table created within a procedure disappears when the procedure exits. See "Temporary Tables" for more information.
- System procedures such as **sp_help** work on temporary table, but only if you use them from *tempdb*.

Setting Options in Procedures

- You can use the **set** command inside a stored procedure. The **set** option remains in effect during the execution of the procedure, then reverts to its former setting.

Getting Information About Procedures

- For a report on the objects referenced by a procedure, use **sp_depends**.
- The *@@error* global variable is zeroed out each time a stored procedure executes successfully.
- To display the text of a **create procedure** statement, use the system procedure **sp_helptext** with the procedure name as the parameter.

- To display the text of a system-defined procedure, execute **sp_helptext** from the *master* database.

Nested Procedures

- Procedure nesting occurs when one stored procedure calls another.
- If you execute a procedure that calls another procedure, the called procedure can access objects created by the calling procedure.
- The nesting level is incremented when the called procedure begins execution and it is decremented when the called procedure completes execution. Exceeding the maximum of 16 levels of nesting causes the transaction to fail.
- The current nesting level is stored in the *@@nestlevel* global variable.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

Permissions

create procedure permission defaults to the Database Owner, who can transfer it to other users.

Permission to use a procedure must be granted explicitly with the **grant** command, and may be revoked with the **revoke** command.

Procedures that perform **select**, **execute**, or data modification statements, grant "implicit permission" to users who have permission to execute the procedure, even if the users don't have permission to perform these operations on the referenced database objects. The procedure and the underlying objects must be owned by the same user.

See Also

Commands	**begin...end, break, continue, declare, drop procedure, execute, go to Label, grant, if...else, return, select, waitfor, while**
Topics	Batch Queries, Comments, Parameters, Variables (Local and Global), Wildcard Characters
System procedures	**sp_depends, sp_helptext, sp_rename**

create rule

Function

Specifies the domain of acceptable values for a particular column or for any column of a user-defined datatype.

Syntax

```
create rule [owner.]rule_name
   as condition_expression
```

Keywords and Options

rule_name – is the name of the new rule. It must conform to the rules for identifiers.

condition_expression – specifies the conditions that define the rule. It can be any expression that is valid in a **where** clause, and can include arithmetic operators, relational operators, **in**, **like**, **between**, etc. However, it cannot reference any column or other database object. Built-in functions that do not reference database objects **can** be included.

A *condition_expression* takes one argument. The argument is prefixed by the "@" symbol, and refers to the value that is entered via the **update** or **insert** command. You can use any name or symbol to represent the value when you write the rule, but the first character must be the "@" symbol. Enclose character and date constants in quotes, and precede binary constants with "0x".

Examples

```
1. create rule limit
   as @advance < $1000

2. create rule pubid_rule
   as @pub_id in ('1389', '0736', '0877')

3. create rule picture
   as @value like '_-%[0-9]'
```

Comments

Restrictions

- You can only create a rule in the current database.

- Rules do not apply to the data already existing in the database at the time the rules are created.
- **create rule** statements cannot be combined with other statements in a single batch.
- You cannot bind a rule to a SQL Server-supplied datatype or to a column of type *text*, *image*, or *timestamp*.
- You must drop a rule before you **create** a new one of the same name, and you must unbind a rule before you drop it. Use:

  sp_unbindrule *objname* [, futureonly]

Binding Rules

- Use the system procedure **sp_bindrule** to bind a rule to a column or user-defined datatype. Its syntax is:

  sp_bindrule *rulename*, *objname* [, futureonly]

- A rule bound to a user-defined datatype is activated when you insert a value into, or update, a column of that type. Rules do **not** test values inserted into variables of that type.
- The rule must be compatible with the datatype of the column. For example, you cannot use:

  @value like A%

  as a rule for an exact or approximate numeric column. If the rule is not compatible with the column to which you've bound it, SQL Server generates an error message when it tries to insert a value (not when you bind it).
- You can bind a rule to a column or datatype without unbinding an existing rule.
- Rules bound to columns always take precedence over rules bound to user-defined datatypes, regardless of which was most recently bound. The following chart indicates the precedence when binding rules to columns and user-defined datatypes where rules already exist:

New Rule bound to	Old rule bound to	
	user-defined datatype	**column**
user-defined datatype	new rule replaces old	no change
column	new rule replaces old	new rule replaces old

Table 1-7: Rule binding precedence

Rules and Nulls

- Rules do not override column definitions. If a rule is bound to a column that allows null values, you can insert NULL into the column, implicitly or explicitly, even though NULL is not included in the text of the rule. For example, if you create a rule specifying "@val in (1,2,3)" or "@amount > 10000", and bind this rule to a table column which allows null values, you can still insert NULL into that column. The column definition overrides the rule.

Getting Information About Rules

- After a rule is bound to a particular column or user-defined datatype, its ID is stored in the *syscolumns* or *systypes* system tables.
- To get a report on a rule, use **sp_help**.
- To display the text of a rule, which is stored in the *syscomments* system table, execute the system procedure **sp_helptext** with the rule name as the parameter.
- To rename a rule, use **sp_rename**.

Defaults and Rules

- If a column has both a default and a rule associated with it, the default must fall within the domain defined by the rule. A default that conflicts with a rule will never be inserted. SQL Server generates an error message each time it attempts to insert the default.

Using Integrity Constraints in Place of Rules

- You can also define rules using **check** integrity constraints with the **create table** statement. However, these constraints are specific for that table; you cannot bind them to other tables. See "**create table**" and "**alter table**" for information about integrity constraints.

Standards and Compliance

Standard	Compliance Level	Comments
SQL89	Transact-SQL extension	To create rules using SQL compliant syntax, use the **check** clause of the **create table** statement.
SQL92	Transact-SQL extension	

Permissions

create rule permission defaults to the Database Owner, who can transfer it to other users.

See Also

Commands	**alter table**, **create default**, **create table**, **drop default**, **drop rule**
Topics	Batch Queries
System procedures	**sp_bindrule**, **sp_help**, **sp_helptext**, **sp_rename**, **sp_unbindrule**

create schema

Function

Creates a new collection of tables, views and permissions for a database user.

Syntax

```
create schema authorization authorization_name
   create_oject_statement
       [ create_object_statement ... ]
   [ permission_statement ... ]
```

Keywords and Options

authorization_name – must be the name of the current user in the database.

create_object_statement– is a **create table** or **create view** statement.

permission_statement– is a **grant** or **revoke** command.

Examples

```
create schema authorization pogo
    create table newtitles (
        title_id tid not null,
        title varchar(30) not null)

    create table newauthors (
        au_id id not null,
        au_lname varchar(40) not null,
        au_fname varchar(20) not null)

    create table newtitleauthors (
        au_id id not null,
        title_id tid not null)

    create view tit_auth_view
    as
        select au_lname, au_fname
            from newtitles, newauthors,
                newtitleauthors
        where
        newtitleauthors.au_id = newauthors.au_id
        and
        newtitleauthors.title_id =
             newtitles.title_id
```

```
grant select on tit_auth_view to public
revoke select on tit_auth_view from churchy
```

Comments

- Schemas can be created only in the current database.
- The schema authorization name must be the name of the current user.
- The user must have the correct command permissions (**create table** and / or **create view**). If the user creates a view on tables owned by another database user, permissions on the view are checked when a user attempts to access data through the view, not when the view is created.
- The **create schema** command is terminated by:
  - The regular command terminator ("go" by default in **isql**).
  - Any statement other than **create table**, **create view**, **grant**, or **revoke**.
- If any of the statements within a **create schema** statement fail, the entire command is rolled back as a unit, and none of the commands take affect.
- **create schema** adds information about tables, views and permissions to the system tables. Use the appropriate drop command (**drop table** or **drop view**) to drop objects created with **create schema.** Permissions granted or revoked in a schema can be changed with the standard **grant** and **revoke** commands outside the schema creation statement.

Standards and Compliance

Standard	Compliance Level
SQL89	Compliant
SQL92	Entry level compliant

Permissions

create schema can be executed by any user of a database; the user must have permission to create the objects specified in the schema, that is, **create table** and / or **create view** permission.

See Also

Commands	**create table, create view, grant, revoke**

create table

Function

Creates new tables and optional integrity constraints.

Syntax

```
create table [database.[owner].]table_name
  (column_name datatype
      [default {constant_expression | user | null}]
      {[{identity | null | not null}]
      | [[constraint constraint_name]
          {{unique | primary key}
              [clustered | nonclustered]
              [with fillfactor = x] [on segment_name]
          | references [[database.]owner.]ref_table
              [(ref_column)]
          | check (search_condition)}]}...

  | [constraint constraint_name]
      {{unique | primary key}
          [clustered | nonclustered]
          (column_name [{, column_name}...])
          [with fillfactor = x] [on segment_name]
      | foreign key (column_name [{, column_name}...])
          references [[database.]owner.]ref_table
              [(ref_column [{, ref_column}...])]
      | check (search_condition)}

  [{, {next_column | next_constraint}}...])

  [on segment_name]
```

Keywords and Options

table_name – is the name of the new table. It must be unique within the database and to the owner. If you have **set quoted_identifier on**, you can use a delimited identifier for the table. Otherwise, it must conform to the rules for identifiers. For more information about valid table names, see ‘‘Identifiers’’.

You can create a temporary table either by preceding the table name in a **create table** statement with a pound sign (#), or by preceding the table name with “*tempdb..*”. The first 13 characters of a temporary table name which begins with a pound sign (including the pound sign) must be unique to a user, per session.

Such tables can be accessed only by the current SQL Server session. They are stored in *tempdb..objects* by their names plus a system-supplied numeric suffix, and they disappear at the end of the current session, or when they are explicitly dropped. Temporary tables created with the "*tempdb..*" prefix are sharable among SQL Server sessions. They exist until they are explicitly dropped by their owner, or until SQL Server reboots. See "Temporary Tables" for more information.

You can create a table in a different database, as long as you are listed in the *sysusers* table and have **create table** permission for that database. For example, to create a table called *newtable* in the database *otherdb*:

```
create table otherdb..newtable ...
```

or:

```
create table otherdb.yourname.newtable ...
```

column_name – is the name of the column in the table. It must be unique in the table. If you have **set quoted_identifier on**, you can use a delimited identifier for the column. Otherwise, it must conform to the rules for identifiers. For more information about valid column names, see ''Identifiers''.

datatype – is the datatype of the column. System or user-defined datatypes are acceptable. Certain datatypes expect a length, *n*, in parentheses:

```
datatype(n)
```

Others expect a precision, *p*, and scale, *s*:

```
datatype(p,s)
```

See "Datatypes" for more information.

default – specifies a default value for a column. If you have declared a default and the user does not provide a value for this column when inserting data, SQL Server inserts this value. The default can be either a constant expression, **user** to insert the name of the user who is performing the insert, or **null** to insert the null value. Defaults declared for columns with the IDENTITY property have no effect on column values.

constant_expression – is a constant expression to use as a default value for the column. It cannot include the name of any columns or other database objects, but you can include built-in functions that

do not reference database objects. This default value must be compatible with the datatype of the column.

user | **null** – specifies that SQL Server should insert the user name or the null value as the default if the user does not supply a value. For **user**, the datatype of the column must be either *char(30)* or *varchar(30*). For **null**, the column must allow null values.

identity – indicates that the column has the IDENTITY property. Each table in a database can have one IDENTITY column with a type of *numeric* and scale zero. IDENTITY columns are not updatable and do not allow nulls.

IDENTITY columns are used to store sequential numbers, such as invoice numbers or employee numbers, that are generated automatically by SQL Server. The value of the IDENTITY column uniquely identifies each row in a table.

null | **not null** – specifies that SQL Server assigns a null value if a user does not provide a value during an insertion and no default exists (for **null**), or that a user must provide a non-null value if no default exists (for **not null**). If you do not specify **null** or **not null**, SQL Server uses **not null** by default. However, you can switch this default using **sp_dboption** to make the default compatible with the ANSI standard.

constraint – introduces the name of an integrity constraint. This keyword and the *constraint_name* are optional.

constraint_name – is the name of the constraint. It must conform to the rules for identifiers and be unique in the database. If you do not specify the name for a referential or check constraint, SQL Server generates a name as follows:

tabname_colname_objectid

where *tabname* is the first 10 characters of the table name, *colname* is the first 5 characters of the column name, and *objectid* is the object ID number for the constraint. If you do not specify the name for a unique or primary key constraint, SQL Server generates a name as follows:

tabname_colname_tabindid

where *tabindid* is a string concatenation of the table ID and index ID.

unique – constrains the values in the indicated column or columns so that no two rows can have the same non-null value. This

constraint creates a unique index that can only be dropped if the constraint is dropped using **alter table**.

primary key – constrains the values in the indicated column or columns so that no two rows can have the same value, and so that the value cannot be NULL. This constraint creates a unique index that can only be dropped if the constraint is dropped using **alter table**.

clustered | **nonclustered** – specifies that the index created by a **unique** or **primary key** constraint is a clustered or nonclustered index. **clustered** is the default for primary key constraints, **nonclustered** is the default for unique constraints. There can be only one clustered index per table. See "**create index**" for more information.

fillfactor – specifies how full SQL Server will make each page when it is creating a new index on existing data. The **fillfactor** percentage is relevant only at the time the index is created. As the data changes, the pages are not maintained at any particular level of fullness.

The default for **fillfactor** is 0; this is used when you do not include **with fillfactor** in the **create index** statement (unless the value has been changed with **sp_configure**). When specifying a **fillfactor**, legal values are between 1 and 100.

A **fillfactor** of 0 creates clustered indexes with completely full pages and nonclustered indexes with completely full leaf pages. It leaves a comfortable amount of space within the index B-tree in both the clustered and nonclustered cases. There is seldom a reason to change the **fillfactor**.

If the **fillfactor** is set to 100, SQL Server creates both clustered and nonclustered indexes with each page 100% full. A **fillfactor** of 100 makes sense only for read-only tables—tables to which no additional data will ever be added.

fillfactor values smaller than 100 (except 0, which is a special case) cause SQL Server to create new indexes with pages that are not completely full. A **fillfactor** of 10 might be a reasonable choice if you are creating an index on a table that will eventually hold a great deal more data, but small **fillfactor** values cause each index (or index and data) to take more storage space.

◆ ***WARNING!***

Creating a clustered index with a fillfactor affects the amount of storage space your data occupies, since SQL Server redistributes the data as it creates the clustered index.

on *segment_name* – specifies that the index is to be created on the named segment. Before the **on** *segment_name* option can be used, the device must be initialized with **disk init**, and the segment must be added to the database with the **sp_addsegment** system procedure. See your System Administrator or use **sp_helpsegment** for a list of the segment names available in your database.

If you specify **clustered** and use the **on** *segment_name* option, the entire table migrates to the segment you specify, since the leaf level of the index contains the actual data pages.

references – specifies a column list for a referential integrity constraint. You can only specify one column value for a column-constraint. By including this constraint with a table that references another table, any data inserted into the "referencing" table must already exist in the "referenced" table.

To use this constraint, you must have **references** permission on the referenced table. The specified columns in the referenced table must be constrained by a unique index (created by either a **unique** constraint or a **create index** statement). If no columns are specified, there must be a **primary key** constraint on the appropriate columns in the referenced table. Also, the datatypes of the referencing table columns must exactly match the datatype of the referenced table columns.

foreign key – specifies that the listed column(s) are foreign keys in this table whose target keys are the columns listed in the following **references** clause. The foreign key syntax is permitted only for table-level constraints, not for column-level constraints.

ref_table – is the name of the table that contains the referenced columns. You can reference tables in another database.

ref_column – is the name of the column or columns in the referenced table.

check – specifies a *search_condition* constraint which SQL Server enforces for all the rows in the table. You can specify **check** constraints as table or column constraints; **create table** allows multiple **check** constraints in a column definition.

search_condition – is the **check** constraint on the column values. These constraints can include:

- A list of constant expressions introduced with **in**

- A set of conditions introduced with **like** which may contain wildcard characters

Column check constraints can reference only the columns on which they are defined; they cannot reference other columns in the table. Table check constraints can reference any columns in the table.

An expression can include arithmetic operators and functions. The *search_condition* cannot contain subqueries, aggregate functions, host variables, or parameters.

next_column | *next_constraint* – indicates that you can include additional column definitions or table-constraints (separated by commas) using the same syntax described for a column definition or table constraint definition.

on *segment_name* – specifies the name of the segment on which to place the table. When using **on** *segment_name* the logical device must already be assigned to the database with **create database** or **alter database**, and the segment must have been created in the database with **sp_addsegment**. See your System Administrator or use **sp_helpsegment** for a list of segment names available in your database.

Examples

```
1. create table titles
   (title_id       tid          not null,
   title           varchar(80)  not null,
   type            char(12)     not null,
   pub_id          char(4)      null,
   price           money        null,
   advance         money        null,
   total_sales     int          null,
   notes           varchar(200) null,
   pubdate         datetime     not null,
   contract        bit          not null)
```

Creates the *titles* table.

```
2. create table “compute”
   (“max” int,”min” int, “total score” int)
```

Creates the *compute* table. The table name and the column names *max* and *min* are enclosed in double quotes because they are reserved words. The *total score* column name is enclosed in double quotes because it contains an embedded blank. Before creating this table, you must **set quoted_identifier on**.

3.
```
create table sales
(stor_id        char(4)       not null,
ord_num         varchar(20)   not null,
date            datetime      not null,
unique clustered (stor_id, ord_num))
```

   Creates the *sales* table and a clustered index in one step with a unique constraint. (In the *pubs2* database installation script, there are separate **create table** and **create index** statements.)

4.
```
create table salesdetail
(stor_id    char(4)                 not null,
ord_num     varchar(20)             not null,
title_id    tid                     not null
            references titles(title_id),
qty         smallint    default 0   not null,
discount    float                   not null,

constraint salesdet_constr
    foreign key (stor_id, ord_num)
    references sales(stor_id, ord_num))
```

   Creates the *salesdetail* table with two referential integrity constraints and one default value. There is a table-level referential integrity constraint named *salesdet_constr*, and a column-level referential integrity constraint on the *title_id* column without a specified name. Both constraints specify columns which have unique indexes in the referenced tables (*titles* and *sales*). The **default** clause with the *qty* column specifies 0 as its default value.

5.
```
create table publishers
(pub_id         char(4)       not null
        check (pub_id in ("1389", "0736", "0877",
                "1622", "1756")
        or pub_id like "99[0-9][0-9]"),
pub_name        varchar(40)   null,
city            varchar(20)   null,
state           char(2)       null)
```

   Creates the table *publishers* with a check constraint on the *pub_id* column. This column-level constraint could be used in place of the *pub_idrule* included in the *pubs2* database:

```
create rule pub_idrule
as @pub_id in ("1389", "0736", "0877",
                "1622", "1756")
or @pub_id like "99[0-9][0-9]"
```

```
6. create table sales_daily
   (stor_id        char(4)       not null,
   ord_num         numeric(10,0) identity,
   ord_amt         money         null)
```

Specifies the *ord_num* column as the IDENTITY column for the *sales_daily* table. The first time you insert a row into the table, SQL Server assigns a value of 1 to the IDENTITY column. On each subsequent insert, the value of the column is incremented by one.

Comments

Restrictions

- There can be up to 2 billion tables per database and 250 columns per table. The number of rows per table is limited only by available storage.
- The maximum number of bytes per row is 1962. If you create tables with *varchar*, *nvarchar*, or *varbinary* columns whose total defined width is greater than 1962 bytes, a warning message appears, but the table is created. If you try to insert more than 1962 bytes into such a row, or to update a row so that its total row size is greater than 1962, SQL Server produces an error message and the command fails.
- The table is created in the currently open database unless you specify a different database in the **create table** statement. You can create a table or index in another database if you are listed in the *sysusers* table and have **create table** permission in the database.
- Space is allocated to tables and indexes in increments of one extent, or 8 pages, at a time. Each time an extent is filled, another extent is allocated. To see the amount of space allocated and used by a table, use **sp_spaceused**.

Column Definitions

- When you create a column from a user-defined datatype:
  - You cannot change the length, precision, or scale.
  - You can use a NULL type to create a NOT NULL column, but not to create an IDENTITY column.
  - You can use a NOT NULL type to create a NULL column, but not to create an IDENTITY column.

- You can use an IDENTITY type to create a NOT NULL column, but the column inherits the IDENTITY property. You cannot use an IDENTITY type to create a NULL column.

- Only columns with variable-length datatypes can store null values. When you create a NULL column with a fixed-length datatype, SQL Server automatically converts it to the corresponding variable-length datatype. SQL Server does not inform the user of the type change.

  The following chart lists the fixed-length datatypes and the variable-length datatypes to which they are converted. Certain variable-length datatypes, such as *moneyn*, are reserved types; you cannot use them to create columns, variables, or parameters.

Original Fixed-Length Datatype	Converted To
char	*varchar*
nchar	*nvarchar*
binary	*varbinary*
datetime	*datetimn*
float	*floatn*
int, *smallint*, and *tinyint*	*intn*
decimal	*decimaln*
numeric	*numericn*
money and *smallmoney*	*moneyn*

Table 1-8: Variable-length datatypes used to store nulls

- For a report on a table and its columns, execute the system procedure **sp_help**.

Temporary Tables

- Temporary tables are stored in the temporary database, *tempdb*.
- You can associate rules, defaults and indexes with temporary tables, but you cannot create a view on temporary tables or associate triggers with them.
- When you create a temporary table, you can only use a user-defined datatype if the type is in *tempdb..systypes*. To add a user-defined datatype to *tempdb* for the current session only, execute **sp_addtype** while using *tempdb*. To add the datatype permanently, execute **sp_addtype** while using *model* and then restart SQL Server so that *model* is copied to *tempdb*.

Using Indexes

- A table "follows" its clustered index. If you create a table on one segment, and then create its clustered index on another segment, the table will "migrate" to the segment where the index is created.
- You can make inserts, updates, and selects faster by creating a table on one segment, and its non-clustered indexes on another segment, if these segments are on separate physical devices. See the *System Administration Guide* for more information.

Renaming a Table or Its Columns

- Use **sp_rename** to rename a table or column.
- After renaming a table or any of its columns, use **sp_depends** to determine which procedures, triggers, and views depend on the table, then redefine these objects.

◆ ***WARNING!***

If you do not redefine these dependent objects, they will no longer work after SQL Server recompiles them.

Defining Integrity Constraints

- The **create table** statement helps control a database's integrity through a series of integrity constraints as defined by the ANSI standard. These integrity constraint clauses restrict the data that users can insert into a table. You can also use defaults, rules, indexes, and triggers to enforce database integrity.

  Integrity constraints offer the advantages of defining integrity controls in one step during the table creation process and of simplifying the process to create those integrity controls. However, integrity constraints are more limited in scope and less comprehensive than defaults, rules, indexes, and triggers.
- You must declare constraints that operate on more than one column as table-level constraints; declare constraints that operate on just one column as column-level constraints. The difference is syntactic: you place column-level constraints after the column name and datatype, before the delimiting comma (see example 4). You enter table-level constraints as separate comma-delimited clauses (see example 2). SQL Server treats table-level and

column-level constraints the same way; neither way is more efficient than the other.

- You can create the following types of constraints at the table level or the column level:
  - a **unique** constraint requires that no two rows in a table have the same non-null values in the specified columns. In addition, a **primary key** constraint requires that there be no null values in the column.
  - referential integrity (**references**) constraints require that data being inserted or updated in specific columns must already have matching data in the specified table and columns.
  - **check** constraints limit the values of the data inserted into the columns.

  You can also enforce data integrity by restricting the use of null values in a column (the **null** or **not null** keywords) and by providing default values for columns (the **default** clause).
- You can define primary, foreign, and common keys on a table with the system procedures **sp_primarykey**, **sp_foreignkey**, and **sp_commonkey**. These procedures save information in system tables which can help clarify the relationships between tables in a database. But they do not replace the functions of the **primary key** and **foreign key** keywords in a **create table** statement. For a report on the keys that have been defined, use **sp_helpkey**. For a report on frequently used joins, execute **sp_helpjoins**.
- Transact-SQL provides several mechanisms for integrity enforcement. In addition to the constraints you can declare as part of **create table**, you can create rules, defaults, indexes, and triggers. The following table summarizes the integrity constraints and describes the other methods of integrity enforcement:

In create table	Other Methods
unique constraint	**create unique index** (on a column that allows null values)
primary key constraint	**create unique index** (on a column that does not allow null values)
references constraint	**create trigger**
check constraint (table level)	**create trigger**

Table 1-9: Methods of integrity enforcement

In create table	Other Methods
check constraint (column level)	**create trigger** or **create rule** and **sp_bindrule**
default clause	**create default** and **sp_bindefault**

Table 1-9: Methods of integrity enforcement (continued)

Which method you choose depends on your requirements. For example, triggers provide more complex handling of referential integrity (such as referencing other columns or objects) than those declared in **create table**. Also, the constraints defined in a **create table** statement are specific for that table. Unlike triggers, rules, or defaults, you cannot bind them to other tables, and you can only drop or change them using **alter table**. Constraints cannot contain subqueries or aggregate functions, even on the same table.

- The **create table** command can include many constraints, with these limitations:
  - The number of **unique** constraints is limited by the number of indexes that table can have.
  - A table can have only one **primary key** constraint.
  - You can include only one **default** clause per column of a table, but you can define different constraints on the same column.

  For example:

```
create table discount_titles
(title_id   varchar(6)  default "PS7777" not null
        unique clustered
        references titles(title_id)
        check (title_id like "PS%"),
new_price   money)
```

  Column *title_id* of the new table *discount_titles* is defined with each integrity constraint.

- You can create error messages and bind them to referential integrity and **check** constraints. Create messages with **sp_addmessage**, and bind them to the constraints with **sp_bindmsg.** For more information, see **sp_addmessage** and **sp_bindmsg** in Volume 2.
- SQL Server always evaluates check constraints before the referential constraints are enforced. Triggers are evaluated after all the integrity constraints. If any constraint fails, SQL Server

cancels the data modification statement and any associated triggers are not executed. However, a constraint violation **does not** roll back the current transaction.

- For information about any constraints defined for a table, use **sp_helpconstraint**. **sp_helpconstraint** is described in Volume 2.

Unique and Primary Key Constraints

- You can declare **unique** constraints at the column level or the table level. **unique** constraints require that all non-null values in the specified columns must be unique. No two rows in the table are allowed to have the same value in the specified column.
- A **primary key** constraint is a more restrictive form of **unique** constraint. Columns with **primary key** constraints may not contain null values.

➤ ***Note***

create table's *unique* and *primary key* constraints create indexes that define unique or primary key attributes of columns. ***sp_primarykey***, ***sp_foreignkey***, and ***sp_commonkey*** define logical relationships between columns. These relationships must be enforced using indexes and triggers.

- Table-level **unique** or **primary** key constraints appear in the **create table** statement as separate items, and must include the names of one or more columns from the table being created.
- **unique** or **primary key** constraints create a unique index on the specified columns. The **unique** constraint in example 2, **`unique clustered (stor_id, ord_num),`** creates a unique, clustered index, exactly the same as the statement:

```
create unique clustered index salesind
    on sales (stor_id, ord_num)
```

  The only difference is the index name, which you could set to *salesind* by naming the constraint.

- **unique** constraints create unique nonclustered indexes by default; **primary key** constraints create unique clustered indexes by default. There can be only one clustered index on a table, so you can specify only one **unique clustered** or **primary key clustered** constraint.

- The **unique** or **primary key** integrity constraints of **create table** offer a simpler alternative to the **create index** statement. They have the following limitations:
  - You cannot create non-unique indexes.
  - You cannot use all the options provided by **create index.**
  - You must drop these indexes using **alter table drop constraint.**

Referential Integrity Constraints

- Referential integrity constraints require that data inserted into a "referencing" table which defines the constraint must have matching values in a "referenced" table. A referential integrity constraint is satisfied for either of the following conditions:
  - The data in the constrained column(s) of the referencing table contains a null value.
  - The data in the constrained column(s) of the referencing table matches data values in the corresponding columns of the referenced table.

  Using the *pubs2* database as an example, a row inserted into the *salesdetail* table (which records the sale of books) must have a valid *title_id* in the *titles* table. *salesdetail* is the referencing table and *titles* table is the referenced table. Currently, *pubs2* enforces this referential integrity using a trigger. However, the *salesdetail* table could include this column definition and referential integrity constraint to accomplish the same task:

  ```
  title_id tid
      references titles(title_id)
  ```
- A table can include a referential integrity constraint on itself.
- You cannot delete rows or update column values from a referenced table that match values in a referencing table.
- You cannot drop the referenced table until the referencing table is dropped or the referential integrity constraint is removed (unless it includes only a referential integrity constraint on itself).
- SQL Server does not enforce referential integrity constraints for temporary tables.
- To create a table that references another user's table, you must have **references** permission on the referenced table. For information about assigning **references** permissions, see the **grant** command.

- Table-level referential integrity constraints appear in the **create table** statement as separate items. They must include the **foreign key** clause and a list of one or more column names.

  Column names in the **references** clause are optional only if the columns in the referenced table are designated as a primary key through a **primary key** constraint.

  The referenced columns must be constrained by a unique index in that referenced table. You can create that unique index using either the **unique** constraint or the **create index** statement.

- The datatypes of the referencing table columns must exactly match the datatypes of the referenced table columns. For example, the datatype of *col1* in the referencing table (*test_type*) matches the datatype of *pub_id* in the referenced table (*publishers*):

```
create table test_type
(col1 char(4) not null
    references publishers(pub_id),
col2 varchar(20) not null)
```

- The referenced table must exist at the time you define the referential integrity constraint. For tables that cross-reference one another, use the **create schema** statement to define both tables "simultaneously." As an alternative, create one table without the constraint and later add it using the **alter table** statement. See **create schema** or **alter table** in this chapter for more information.

- The **create table** referential integrity constraints offer a simple way to enforce data integrity. Unlike triggers, they **cannot**:

  - "Cascade" changes through related tables in the database
  - Enforce complex restrictions by referencing other columns or database objects
  - Perform "what-if" analysis

  Referential integrity constraints do not roll back transactions when a data modification violates the constraint. Triggers allow you to choose whether to roll back or continue the transaction depending on how you handle referential integrity.

➤ ***Note***

SQL Server checks referential integrity constraints before any triggers, so a data modification statement that violates the constraint does not also fire the trigger.

Using Cross-Database Referential Integrity Constraints

- When you create a cross-database constraint, SQL Server stores the following information in the *sysreferences* system table of each database:

Information	*sysreferences* Column for	
	Referenced Table	Referencing Table
Key Column IDs	*refkey1* through *refkey16*	*fokey1* through *fokey16*
Table ID	*reftabid*	*tableid*
Database Name	*pmrydbname*	*frgndbname*

Table 1-10: Information stored about cross-database referential integrity constraints

- You can drop the referencing table or its database without problems. SQL Server automatically removes the foreign key information from the referenced database.
- Because the referencing table depends on information from the referenced table, SQL Server does not allow you to:
  - Drop the referenced table,
  - Drop the external database that contains it, or
  - Rename either database with **sp_renamedb**

  You must first remove the cross-database constraint with **alter table.**
- Each time you add or remove a cross-database constraint, or drop a table that contains a cross-database constraint, dump **both** of the affected databases.

◆ ***WARNING!***

Loading earlier dumps of these databases could cause database corruption.

- The *sysreferences* system table stores the **name**—not the ID number—of the external database. SQL Server cannot guarantee referential integrity if you use **load database** to change the database name or to load it onto a different server.

◆ ***WARNING!***

Before dumping a database in order to load it with a different name or move it to another SQL Server, use alter table to drop all external referential integrity constraints.

Check Constraints

- **check** constraints limit the values users can insert into a column of a table. A **check** constraint specifies a *search_condition* which any value must pass before it is inserted into the table. A *search_condition* can include:
  - A list of constant expressions introduced with **in**
  - A range of constant expressions introduced with **between**
  - A set of conditions introduced with **like** which may contain wildcard characters

  An expression can include arithmetic operators and Transact-SQL built-in functions. The *search_condition* cannot contain subqueries, aggregate functions, or a host variable or parameter. SQL Server does not enforce **check** constraints for temporary tables.
- If the check constraint is a column-level check constraint, it can only reference the column on which it is defined; it cannot reference other columns in the table.
- Table-level check constraints can reference any columns in the table.
- **create table** allows multiple **check** constraints in a column definition.
- **check** integrity constraints offer an alternative to using rules and triggers. They are specific to the table in which they are created, and cannot be bound to columns of other tables or to user-defined datatypes.

IDENTITY Columns

- The first time you insert a row into the table, SQL Server assigns the IDENTITY column a value of 1. Each new row gets a column value one higher than the last. This value takes precedence over any defaults declared for the column in the **create table** statement or bound to the column with the **sp_bindefault** system procedure.

- The maximum value that can be inserted into the IDENTITY column is $10^{precision} - 1$.
- Only the table owner, Database Owner, or System Administrator can explicitly insert a value into an IDENTITY column after setting **identity_insert on** for the base table. (At any given time, a user can turn the **identity_insert** option on for a single table in a database.)

  Inserting a value into the IDENTITY column allows you to specify a "seed" value for the column or to restore a row that was deleted in error. Unless you have created a unique index on the IDENTITY column, SQL Server does not verify the uniqueness of the value; you can insert any positive integer.
- You can reference an IDENTITY column using the **syb_identity** keyword, qualified by the table name where necessary, in place of the actual column name.
- System Administrators can use the **auto identity** database option to automatically include a 10-digit IDENTITY column in new tables. To turn this feature on in a database, use:

  sp_dboption *database_name*, "auto identity", "true"

  Each time a user creates a table in the database without specifying either a **primary** key, a **unique** constraint, or an IDENTITY column, SQL Server automatically defines an IDENTITY column. This column, **SYB_IDENTITY_COL**, is not visible when you retrieve columns with the **select** * statement; you must explicitly include the column name in the select list.
- Server failures can create gaps in IDENTITY column values. The maximum size of the gap depends on the setting of the **identity burning set factor** configuration variable. Gaps can also occur due to transaction rollbacks, the deletion of rows, or the manual insertion of data into the IDENTITY column.

Standards and Compliance

Standard	Compliance Level	Comments
SQL89	Compliant	The following are Transact-SQL extensions: • The **on segment_name** clause • Use of a database name to qualify a table or column name • IDENTITY columns • The **not null** column default See "Datatypes" for datatype compliance information.
SQL92	Entry level compliant	

Permissions

create table permission defaults to the Database Owner, who can transfer it to other users. Any user can create temporary tables.

See Also

Commands	**alter table, create index, create rule, create schema, create view, drop index, drop rule, drop table**
Topics	Datatypes, IDENTITY Columns
System procedures	**sp_addmessage, sp_addsegment, sp_addtype, sp_bindmsg, sp_commonkey, sp_depends, sp_foreignkey, sp_help, sp_helpjoins, sp_helpsegment, sp_primarykey, sp_rename, sp_spaceused**

create trigger

Function

Creates a trigger, a type of stored procedure often used for enforcing integrity constraints. A trigger executes automatically when a user attempts a specified data modification statement on a specified table.

Syntax

```
create trigger [owner.]trigger_name
   on [owner.]table_name
   {for {insert , update , delete}
   as SQL_statements
```

Or, using the **if update** clause:

```
create trigger [owner.]trigger_name
   on [owner.]table_name
   for {insert , update}
   as
       [if update (column_name)
           [{and | or} update (column_name)]...]
           SQL_statements
       [if update (column_name)
           [{and | or} update (column_name)]...
           SQL_statements]...
```

Keywords and Options

trigger_name – is the name of the trigger. It must conform to the rules for identifiers, and be unique in the database.

insert, **update**, **delete** – can be included in any combination. **delete** cannot be used with the **if update** clause.

SQL_statements – specify trigger conditions and trigger actions. Trigger conditions determine whether the attempted **insert**, **update**, or **delete** cause the trigger actions to be carried out. They often include a subquery preceded by the keyword **if**. In Example 2, the subquery that follows the keyword **if** is the trigger condition.

Trigger actions take effect when the user action (**insert**, **update**, or **delete**) is attempted. If multiple trigger actions are specified, they are grouped with **begin** and **end**.

See "Triggers and Transactions" for a list of statements that are not allowed in a trigger definition. See "The deleted and inserted

Logical Tables'' for information about the *deleted* and *inserted* logical tables that can be included in trigger definitions.

if update is used to test if the specified column has been modified in any way. This allows specified trigger actions to be associated with updates to specified columns. See Example 3. More than one column can be specified, and you can use more than one **if update** statement in a **create trigger** statement.

Examples

1.
```
create trigger reminder
on titles
for insert, update as
print "Don't forget to print a report for
accounting."
```

Prints a message every time anyone tries to add data or change data in the *titles* table.

2.
```
create trigger t1
on titleauthor
for insert as
if (select count(*)
    from titles, inserted
    where titles.title_id = inserted.title_id) = 0
begin
print "Please put the book's title_id in the
    titles table first."
rollback transaction
end
```

Prevents insertion of a new row into *titleauthor* if there is no corresponding *title_id* in the *titles* table.

3.
```
create trigger t2
on publishers
for update as
if update (pub_id) and @@rowcount = 1
begin
    update titles
    set titles.pub_id = inserted.pub_id
    from titles, deleted, inserted
    where deleted.pub_id = titles.pub_id
end
```

If the *pub_id* column of the *publishers* table is changed, make the corresponding change in the *titles* table.

```
4. create trigger t3
   on titleauthor
   for delete as
   begin
       delete titles
       from titles, deleted
       where deleted.title_id = titles.title_id
       delete titleauthor
       from titleauthor, deleted
       where deleted.title_id = titleauthor.title_id
       print "All references to this title have been
       deleted from titles and titleauthor."
   end
```

If any row is deleted from *titleauthor*, that title is also deleted from the *titles* table. If the book was written by more than one author, other references to it in *titleauthor* are also deleted.

Comments

Triggers and Referential Integrity

- Triggers are commonly used to enforce **referential integrity** (integrity rules about relationships between the primary and foreign keys of tables or views), to supply cascading deletes, and to supply cascading updates. (See Examples 2, 3, and 4, respectively.)
- You can also enforce referential integrity using constraints defined with the **create table** statement as an alternative to using **create trigger**. See "**create table**" and "**alter table**" for information about integrity constraints.

The deleted and inserted Logical Tables

- *deleted* and *inserted* are logical (conceptual) tables. They are structurally like the table for which the trigger is defined—that is, the table on which the user action is attempted—and hold the old values or new values of the rows that would be changed by the user action.
- *deleted* and *inserted* tables can be examined by the trigger to determine whether or how the trigger action(s) should be carried out, but cannot be altered themselves by the triggers actions.
- *deleted* tables are used with **delete** and **update**; *inserted* with **insert** and **update**. (An update is a delete followed by an insert: it affects the *deleted* table first, then the *inserted* table).

Trigger Restrictions

- You can create a trigger only in the current database. If you use an owner name to qualify a trigger, you must explicitly qualify the table name the same way. A trigger may reference objects outside the current database.
- A trigger cannot apply to more than one table. However, the same trigger action can be defined for more than one user action (e.g., **insert** and **update**) in the same **create trigger** statement. A table can have a maximum of three different triggers—one each for **insert**, **update**, and **delete**.
- Each new trigger on a table or column for the same operation (**insert**, **update**, or **delete**) overwrites the previous one, and no warning message is given before the overwrite occurs.
- You cannot create a trigger on a temporary table.
- You cannot create a trigger on a view.
- It is recommended that a trigger not include **select** statements that return results to the user, since special handling for these returned results would have to be written into every application program in which modifications to the trigger table are allowed.
- If a trigger references table names, column names, or view names that are not valid identifiers, you must **set quoted_identifier on** before the **create trigger** command and enclose each such name in double quotes. The **quoted_identifier** option does **not** need to be on when the trigger fires.

Getting Information about Triggers

- The execution plan for a trigger is stored in *sysprocedures*. Each trigger is assigned an identification number, which is stored as a new row in *sysobjects* and as an entry in the *sysobjects* row for the table to which it applies.
- To display the text of a trigger, which is stored in *syscomments*, execute the system procedure **sp_helptext**.
- For a report on a trigger, execute the system procedure **sp_help**.
- For a report on the tables and views that are referenced by a trigger, use **sp_depends**.

Triggers and Performance

- In performance terms, trigger overhead is usually very low. The time involved in running a trigger is spent mostly in referencing other tables, which are either in memory or on the database device.
- The *deleted* and *inserted* tables often referenced by triggers are always in memory rather than on the database device, because they are logical tables. The location of other tables referenced by the trigger determines the amount of time the operation takes.

Setting Options Within Triggers

- You can use the **set** command inside a trigger. The **set** option you invoke remains in effect during the execution of the trigger, and then reverts to its former setting. In particular, the **self_recursion** option, can be used inside a trigger, so that data modifications by the trigger itself can cause the trigger to fire again.

Dropping a Trigger

- You must drop and recreate the trigger if you rename any of the objects that the trigger references. You can rename a trigger with **sp_rename**.
- When you drop a table, any triggers associated with it are also dropped.

Actions That Cause Triggers to Fire

- A **truncate table** command is not caught by a **delete** trigger. Although a **truncate table** statement is, in effect, like a **delete** without a **where** clause (it removes all rows), changes to the data rows are not logged, and so cannot "fire" a trigger.

  Since permission for the **truncate table** command defaults to the table owner and is not transferable, only the table owner need worry about inadvertently circumventing a **delete** trigger with a **truncate table** statement.
- The **writetext** command, whether logged or unlogged, does not cause a trigger to fire.
- A trigger fires only once per data modification statement. A complex query containing a **while** loop may repeat an **update** or **insert** many times, and the trigger is fired each time.

Triggers and Transactions

- Once a trigger is defined, the action it specifies on the table to which it applies is always implicitly part of a transaction, along with the trigger itself. Triggers are often used to roll back an entire transaction if an error is detected, or can roll back the effects of a specific data modification:
  - When the trigger contains the **rollback transaction** command, the rollback aborts the entire batch, and any statements in the batch following the rollback will not be executed.
  - When the trigger contains the **rollback trigger**, the rollback affects only the data modification that caused the trigger to fire. The **rollback trigger** command can include a **raiserror** statement. Subsequent statements in the batch are executed.
- Since triggers execute as part of a transaction, the following statements are not allowed in a trigger:
  - All **create** commands, including **create database**, **create table**, **create index**, **create procedure**, **create default**, **create rule**, **create trigger**, and **create view**
  - All **drop** commands
  - **alter table** and **alter database**
  - **truncate table**
  - **grant** and **revoke**
  - **update statistics**
  - **reconfigure**
  - **load database** and **load transaction**
  - **disk init**, **disk mirror**, **disk refit**, **disk reinit**, **disk remirror**, **disk unmirror**
  - **select into**
- If a desired result (such as a summary value) depends on the number of rows a data modification affects, you should use *@@rowcount* to test for multi-row data modifications (an **insert**, **delete**, or **update** based on a **select** statement), and take appropriate actions. Any Transact-SQL statement which does not return rows (such as an **if** statement) sets *@@rowcount* to zero, so the test of *@@rowcount* should occur at the beginning of the trigger.

Update and Insert Triggers

- When an **insert** or **update** command is executed, rows are added to the trigger table and to *inserted* at the same time. The rows in *inserted* are always duplicates of one or more rows in the trigger table.
- An **update** or **insert** trigger can use the **if update** command to determine whether the **update** or **insert** changed a particular column. **if update**(*column_name*) is true for an **insert** statement whenever the column is assigned a value in the select list or in the **values** clause. An explicit NULL or a default assigns a value to a column, and thus activates the trigger. An implicit NULL, however, does not. Here are some examples:

```
1. create table junk
   (a int null,
   b int not null)

2. /* If update is true for either column */
   insert junk (a, b)
   values (1, 2)

3. insert junk
   /* if update is true for either column */
   values(1,2)

4. insert junk /* explicit null */
   /* if update is true for either column */
   values(NULL,2)

5. insert junk (b)/* with a default for column a,
   values(2) /* if update is true for either column */

6. insert junk (b) /* with no default for column a,
   values(2) /* if update is not true for column a */
```

if update is never true for a **delete** statement.

Nesting Triggers and Trigger Recursion

- SQL Server allows nested triggers by default. To prevent triggers from nesting, use **sp_configure** to set the **nested triggers** option to 0 (off), as follows:

```
sp_configure "nested triggers", 0
```

- Triggers can be nested to a depth of 16 levels. If a trigger changes a table on which there is another trigger, the second trigger will fire, and can then call a third trigger, and so forth. If any trigger in the chain sets off an infinite loop, the nesting level will be

exceeded and the trigger will abort, rolling back the transaction which contains the trigger query.

➤ ***Note***

Since triggers are put into a transaction, a failure at any level of a set of nested triggers cancels the entire transaction: all data modifications are rolled back. Supply your triggers with messages and other error handling and debugging aids in order to determine where the failure occurred.

- The global variable *@@nestlevel* contains the nesting level of the current execution. Each time a stored procedure or trigger calls another stored procedure or trigger, the nesting level is incremented. If the maximum of 16 is exceeded, the transaction aborts.
- If a trigger calls a stored procedure that performs actions which would cause the trigger to fire again, the trigger is only re-activated if nested triggers are enabled. Unless there are conditions within the trigger that limit the number of recursions, this causes a nesting-level overflow.

  For example, if an update trigger calls a stored procedure that performs an update, the trigger and stored procedure execute exactly once if **nested triggers** is set off. If **nested triggers** is set on, and the number of updates is not limited by some condition in the trigger or procedure, the procedure/trigger loop continues until it exceeds the 16-level maximum nesting value.
- By default, a trigger does not call itself in response to a second data modification to the same table within the trigger, regardless of the setting of the **nested triggers** configuration variable. A **set** option, **self_recursion**, enables a trigger to fire again as a result of a data modification within the trigger. For example, if an update trigger on one column of a table results in an update to another column, the update trigger only fires once when self recursion is disabled, but it can fire up to 16 times if self recursion is set on. The **nested triggers** configuration variable must also be enabled in order for self recursion to take place.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

Permissions

create trigger permission defaults to the table owner and is not transferable, except to the Database Owner, who can impersonate the table owner by running the **setuser** command.

See Also

Commands	**alter table, create procedure, create table, drop trigger, rollback trigger, set**
System procedures	**sp_commonkey, sp_configure, sp_depends, sp_foreignkey, sp_help, sp_helptext, sp_primarykey, sp_rename, sp_spaceused**

create view

Function

Creates a view, which is an alternative way of looking at the data in one or more tables.

Syntax

```
create view [owner.]view_name
   [(column_name [, column_name]...)]
   as select [distinct] select_statement
   [with check option]
```

Keywords and Options

view_name – is the name of the view. The name cannot include the database name. If you have **set quoted_identifier on**, you can use a delimited identifier. Otherwise, the view name must conform to the rules for identifiers. For more information about valid view names, see ‘‘Identifiers’’.

column_name – specifies names to be used as headings for the columns in the view. If you have set **quoted_identifier on**, you can use a delimited identifier. Otherwise, the column name must conform to the rules for identifiers. For more information about valid column names, see ‘‘Identifiers’’

It is always legal to supply column names, but is required only when a column is derived from an arithmetic expression, function, string concatenation, or constant; when two or more columns have the same name (usually because of a join); or when you want to give a column in a view a different name than the column from which it is derived (see Example 3). Column names can also be assigned in the **select** statement (see Example 4). If no column names are specified, the view columns acquire the same names as the columns in the **select** statement.

select - begins the **select** statement that defines the view.

distinct - specifies that the view may not contain duplicate rows (optional).

select_statement – completes the **select** statement that defines the view. It can use more than one table and other views.

with check option– indicates that all data modification statements are validated against the view selection criteria. All rows inserted or updated through the view must remain visible through the view.

Examples

1. ```
   create view titles_view
   as select title, type, price, pubdate
   from titles
   ```

2. ```
   create view "new view" ("column 1", "column 2")
   as select col1, col2 from "old view"
   ```

   Creates the *new view* view from *old view*. Both columns are renamed in the new view. All view and column names that include embedded blanks are enclosed in double quotation marks. Before creating the view, you must **set quoted_identifier on**.

3. ```
   create view accounts (title, advance, amt_due)
   as select title, advance, price * total_sales
   from titles
   where price > $5
   ```

4. ```
   create view cities
   (authorname, acity, publishername, pcity)
   as select au_lname, authors.city, pub_name,
   publishers.city
   from authors, publishers
   where authors.city = publishers.city
   ```

   Creates a view derived from two base tables, *authors* and *publishers*. The view contains the names and cities of the authors that live in the same city as a publisher.

5. ```
   create view cities2
   as select authorname = au_lname,
   acity = authors.city, publishername = pub_name,
   pcity = publishers.city
   from authors, publishers
   where authors.city = publishers.city
   ```

   Creates a view with the same definition as in Example 3, but with column headings provided in the **select** statement.

6. ```
   create view author_codes
   as select distinct au_id
   from titleauthor
   ```

   Creates a view, *author_codes*, derived from *titleauthor* which lists the unique author identification codes.

```
7. create view price_list (price)
   as select distinct price
   from titles
```

```
8. create view stores_cal
   as select * from stores
   where state = "CA"
   with check option
```

Creates a view of the *stores* table that excludes information about stores outside of California. The **with check option** clause validates each inserted or updated row against the view's selection criteria. Rows for which *state* has any value other than "CA" are rejected.

```
9. create view stores_cal30
   as select * from stores_cal
   where payterms = "Net 30"
```

Creates a view, *stores_cal30*, which is derived from *stores_cal*. The new view inherits the check option from *stores_cal*; all rows inserted or updated through *stores_cal30* must have a *state* value of "CA". Because *stores_cal30* has no **with check option** clause, it is possible to insert or update rows through *stores_cal30* for which *payterms* has a value other than "Net 30".

```
10.create view stores_cal30_check
   as select * from stores_cal
   where payterms = "Net 30"
   with check option
```

Creates a view, *stores_cal30_check*, derived from *stores_cal*. The new view inherits the check option from *stores_cal*. It also has a **with check option** clause of its own. Each row inserted or updated through *stores_cal30_check* is validated against the selection criteria of both *stores_cal* and *stores_cal30_check*. Rows with a *state* value other than "CA" or a *payterms* value other than "Net 30" are rejected.

Comments

- You can use views as security mechanisms by granting permission on a view but not on its underlying tables.

Restrictions on Views

- You can only create a view in the current database.
- The number of columns referenced by a view cannot exceed 250.
- You cannot create a view on a temporary table.

- You cannot create a trigger or build an index on a view.
- You cannot use **readtext** or **writetext** on *text* or *image* columns in views.
- You cannot include **order by** or **compute** clauses, the keyword **into**, or the **union** operator in the **select** statements that define views.
- **create view** statements can be combined with other SQL statements in a single batch.

View Resolution

- If you alter the structure of a view's underlying table(s) by adding or deleting columns, the new columns will not appear in a view defined with a "**select** *" clause unless the view is deleted and redefined. The asterisk shorthand is interpreted and expanded when the view is first created.
- To get a report of the tables or views on which a view depends, and of objects that depend on a view, execute the system procedure **sp_depends**.
- If a view depends on a table (or view) that has been dropped, SQL Server produces an error message if anyone tries to use the view. If a new table (or view) with the same name and schema is created to replace the one that has been dropped, the view again becomes usable.
- You can redefine a view without redefining other views that depend on it unless the redefinition makes it impossible for SQL Server to translate the dependent view.
- To display the text of a view, which is stored in *syscomments*, execute the system procedure **sp_helptext** with the view name as the parameter.
- You can rename a view with **sp_rename**.
- If you define a view with an outer join, and then query the view with a qualification on a column from the inner table of the outer join, the results may be other than what you expect. All rows from the inner table are returned. Rows that do not meet the qualification show a NULL value in the appropriate columns of those rows.
- When you query through a view, SQL Server checks to make sure that all the database objects referenced anywhere in the statement exist, that they are valid in the context of the statement, and that data update commands do not violate data integrity rules. If any

of these checks fail, you get an error message. If the checks are successful, it "translates" the view into an action on the underlying table(s).

- For more information about views, see the *Transact-SQL User's Guide*.

Modifying Data Through Views

- **delete** statements are not allowed on multi-table views.
- **insert** statements are not allowed unless all **not null** columns in the underlying table or view are included in the view through which you are inserting new rows. (SQL Server has no way to supply values for **not null** columns in the underlying table or view.)
- You cannot insert a row through a view that includes a computed column.
- **insert** statements are not allowed on join views created with **distinct** or **with check option.**
- **update** statements are allowed on join views **with check option**. The update fails if any of the affected columns appears in the **where** clause, in an expression that includes columns from more than one table.
- If you insert or update a row through a join view, all affected columns must belong to the same base table.
- You cannot update or insert into a view defined with the **distinct** clause.
- Data update statements cannot change any column in a view that is a computation, and cannot change a view that includes aggregates.

IDENTITY Columns and Views

- You cannot add a new IDENTITY column to a view with the *column_name* = **identity**(*precision*) syntax.
- To insert an explicit value into an IDENTITY column, the table owner, Database Owner, or the System Administrator must **set identity_insert on** for the column's base table, not through the view through which it is being inserted. At any given time, a user can **set identity_insert on** for a single table in a database.

group by *Clauses and Views*

- If a view definition statement includes aggregate functions as well as qualifications and / or a **group by**, that **group by** or qualification is ignored if the query on the view does not include the aggregated column(s).
- Do not define a view with a **group by** unless the view definition also includes an aggregate function. If you do so, the **group by** is ignored when you retrieve data from the view.

distinct *Clauses and Views*

- The **distinct** clause defines a view as a database object that contains no duplicate rows. A row is defined to be a duplicate of another row if all of its columns' values exactly match the same columns' values in another row. Null values are considered to be duplicates of other null values.

  Querying a subset of a view's columns can result in what appear to be duplicate rows. However, the underlying rows in the view are still unique. This is because SQL Server applies the **distinct** requirement to the view's definition when it accesses the view for the first time (before it does any projection and selection), so that all the view's rows are distinct from each other. If you select a subset of columns, some of which contain the same values, the results appear to contain duplicate rows.

  You can specify **distinct** more than once in the view definition's **select** statement to eliminate duplicate rows, as part of an aggregate function or a **group by** clause. For example:

  ```
  select distinct count(distinct title_id), price
  from titles
  ```

- The scope of the **distinct** applies only for that view; it does not cover any new views derived from the **distinct** view.

with check option *Clauses and Views*

- If a view is created **with check option**, each row that is inserted or updated through the view must meet the selection criteria of the view.
- If a view is created **with check option**, all views derived from the "base" view must satisfy its check option. Each row inserted or updated through the derived view must remain visible through the base view.

Standards and Compliance

Standard	Compliance Level	Comments
SQL89	Compliant	The use of more than one **distinct** keyword and the use of "*column_heading = column_name*" in the **select** list are Transact-SQL extensions.
SQL92	Entry level compliant	

Permissions

create view permission defaults to the Database Owner, who can transfer it to other users. The creator of a view must have **select** permission on any objects referenced in the view definition.

See Also

Commands	**create schema, drop view, update**
System procedures	**sp_depends, sp_help, sp_helptext, sp_rename**

dbcc

Function

Database Consistency Checker (**dbcc**) checks the logical and physical consistency of a database. **dbcc** should be used regularly as a periodic check, or if damage is suspected.

Syntax

```
dbcc
   {checktable({table_name|table_id}[, skip_ncindex])
   |
    checkdb [(database_name [, skip_ncindex] )] |

    checkalloc [(database_name [, fix | nofix])] |
    tablealloc ({table_name | table_id}
       [, {full | optimized | fast | null}
       [, fix | nofix ] ]] )|
    indexalloc ({table_name | table_id}, index_id
       [, {full | optimized | fast | null}
       [, fix | nofix ]] ) |

    checkcatalog [(database_name)] |

    dbrepair (database_name, dropdb) |

    reindex ({table_name | table_id}) |
    fix_text ({table_name | table_id}) }
```

Keywords and Options

checktable – checks the specified table to see that index and data pages are correctly linked, that indexes are in properly sorted order, that all pointers are consistent, that the data information on each page is reasonable, and that page offsets are reasonable. If the log segment is on its own device, running **dbcc checktable** on the *syslogs* table reports the log(s) used and free space. For example:

```
Checking syslogs
The total number of data pages in this table is 1.
*** NOTICE: Space used on the log segment is 0.20 Mbytes, 0.13%.
*** NOTICE: Space free on the log segment is 153.4 Mbytes,
99.87%.
DBCC execution completed.  If dbcc printed error messages, see
your System Administrator.
```

If the log segment is not on its own device, this message appears:

```
*** NOTICE:  Notification of log space used/free cannot be
reported because the log segment is not on its own device.
```

The **skip_ncindex** option causes **dbcc checktable** to skip checking the nonclustered indexes on user tables. The default is to check all indexes.

checkdb – runs the same checks as **checktable**, but on each table, including *syslogs*, in the specified database. If no database name is given, **checkdb** checks the current database.

The **skip_ncindex** option causes **dbcc checktable** to skip checking the nonclustered indexes on a user table. The default is to check all indexes.

checkalloc – checks the specified database to see that all pages are correctly allocated, and that no page is allocated that isn't used. If no database name is given, **checkalloc** checks the current database. It always uses the optimized report option (see **tablealloc** below).

checkalloc reports on the amount of space allocated and used. The default mode for **checkalloc** is **nofix**. You must put the database into single-user mode in order to use the **fix** option.

For a discussion of page allocation in SQL Server, see Chapter 11, ''Diagnosing System Problems,'' in the *System Administration Guide*.

tablealloc – checks the specified table to see that all pages are correctly allocated, and that no page is allocated that is not used. This is a smaller version of **checkalloc**, providing the same integrity checks on an individual table. It can be used with the table name or the table's object id (the *id* column from *sysobjects)*.For an example of **tablealloc** output, see Chapter 11, ''Diagnosing System Problems'' in the *System Administration Guide*.

There are three types of reports that can be generated with **tablealloc**: **full**, **optimized**, and **fast**. The **full** option is equivalent to **checkalloc** at a table level; it reports all types of allocation errors. The **optimized** option produces a report based on the allocation pages listed in the object allocation map (OAM) pages for the table. It does not report and cannot fix unreferenced extents on allocation pages not listed in the OAM pages. The **optimized** option is the default if no type is indicated, or if you use **null**.The **fast** option does not produce an allocation report but does produce an exception report of pages that are referenced but not allocated in the extent (2521 errors).

fix | **nofix** – determines whether or not **tablealloc** fixes the allocation errors found in the table. The default is **fix** for all tables except system tables, for which the default is **nofix.** To use the **fix** option with system tables, you must first put the database into single user mode.

➤ ***Note***

You can specify **fix** or **nofix** only if you include a value for the type of report (**full**, **optimized**, **fast**, or **null**).

indexalloc – checks the specified index to see that all pages are correctly allocated, and that no page is allocated that is not used. This is a smaller version of **checkalloc**, providing the same integrity checks on an individual index. It can be used with the table name or the table's object id (the *id* column from *sysobjects)* plus the index's *indid* from *sysindexes*.

indexalloc produces the same three types of reports as **tablealloc**: **full**, **optimized**, and **fast** (see preceding paragraphs on **tablealloc**). The **fix|nofix** option functions the same with **indexalloc** as with **tablealloc**.

➤ ***Note***

You can specify **fix** or **nofix** only if you include a value for the type of report (**full**, **optimized**, **fast**, or **null**).

checkcatalog – checks for consistency in and between system tables. For example, it makes sure that every type in *syscolumns* has a matching entry in *systypes*, that every table and view in *sysobjects* has at least one column in *syscolumns*, and that the last checkpoint in *syslogs* is valid. **checkcatalog** also reports on any segments that have been defined. If no database name is given, **checkcatalog** checks the current database.

dbrepair (*database_name*, **dropdb**) – drops a damaged database. The **drop database** command does not work on a damaged database.

Users cannot be using the database being dropped when this **dbcc** statement is issued (including the user issuing the statement).

reindex – checks the integrity of indexes on user tables by running a "fast" version of **dbcc checktable.** It can be used with either a table's name or *id* (object ID) from *sysobjects*. The function prints a

message when it discovers the first index-related error, then drops and recreates the suspect indexes. It must be run by the System Administrator or table owner after SQL Server's sort order is changed and indexes are marked suspect by SQL Server.

Here's an example:

```
dbcc reindex(titles)

One or more indexes are corrupt. They will be
     rebuilt.
```

In the above example, **dbcc reindex** has discovered one or more corrupt indexes in the table *titles*. The **dbcc** utility next drops and recreates the appropriate indexes. If the indexes for a table are already correct, or if there are no indexes for the table, **dbcc reindex** does not rebuild the index. It prints an informational message instead. The command is also aborted if a table is suspected of containing corrupt data. If that happens, an error message appears instructing the user to run **dbcc checktable**. **dbcc reindex** does not allow re-indexing of system tables. System indexes are checked and rebuilt, if necessary, as an automatic part of recovery after SQL Server is restarted following a sort order change.

fix_text – upgrades *text* values after a SQL Server's character set has been changed from any character set to a new multibyte character set.

Changing to a multibyte character set makes the internal management of *text* data more complicated. Since a *text* value can be large enough to cover several pages, SQL Server must be able to handle characters that span page boundaries. To do so, the server requires additional information on each of the *text* pages. The System Administrator or table owner must run **dbcc fix_text** on each table that has *text* data to calculate the new values needed. See the *System Administration Guide* for more information.

Examples

1. **dbcc checktable(titleauthor)**
2. **dbcc checkalloc(pubs2)**
3. **dbcc tablealloc(publishers, null, nofix)**

   SQL Server returns an optimized report of allocation for this table but does not fix any allocation errors it finds.

4. **dbcc indexalloc ("pubs..titleauthor", 2, full)**

   SQL Server returns a full report of allocation for the index *audidind* with an *indid* of 2 on the *titleauthor* table and fixes any allocation errors it finds.

5. **dbcc dbrepair(pubs2, dropdb)**
6. **dbcc reindex(titles)**
7. **dbcc fix_text(texttest)**

Comments

- **dbcc** (Database Consistency Checker) can be run while the database is active, except for the **dbrepair**(*database_name*, **dropdb**) option and **dbcc checkalloc** with the **fix** option.
- **dbcc** locks database objects as it checks them. See the **dbcc** discussion in the *System Administration Guide* for information on minimizing performance problems while using **dbcc.**
- To qualify a table or index name with a user name and / or the database name, enclose the qualified name in single or double quotation marks. For example:

  dbcc tablealloc("pubs2.pogo.testtable")

- **dbcc reindex** cannot be run within a user-defined transaction.
- **dbcc fix_text** can generate a large number of log records, and it is possible for the log to fill up while running the utility. The utility is designed so that updates are done in a series of small transactions: in case of a log space failure, only a small amount of work is lost. If you run out of log space, clear out your log and restart **dbcc fix_text** using the same table being upgraded when the original **dbcc fix_text** failed.
- If you attempt to use **select**, **readtext**, or **writetext** on *text* values after changing to a multibyte character set and have not run **dbcc fix_text**, the command fails and an error message is generated instructing you to run **dbcc fix_text** on the table. However, you can delete *text* rows after changing character sets without running **dbcc fix_text.**
- **dbcc** output is sent as messages and / or errors, rather than as result rows. Client programs and scripts should check the appropriate error handlers.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

Permissions

Only the table owner can execute **dbcc** with the **checktable**, **fix_text**, and **reindex** options. Only the Database Owner can use the **checkdb**, **checkalloc**, **checkcatalog**, **dbrepair**, **indexalloc**, and **tablealloc** options.

See Also

Commands	**drop database, reconfigure**
System procedures	**sp_configure, sp_helpdb**

deallocate cursor

Function

Makes a cursor inaccessible and releases all memory resources committed to that cursor.

Syntax

```
deallocate cursor cursor_name
```

Parameters

cursor_name – is the name of the cursor to deallocate.

Examples

```
deallocate cursor authors_crsr
```

Deallocates the cursor named *authors_crsr*.

Comments

- SQL Server returns an error message if the cursor does not exist.
- You must deallocate a cursor before you can use its cursor name as part of another **declare cursor** statement.
- **deallocate cursor** has no effect on memory resource usage when specified in a stored procedure or trigger.
- You can **deallocate** a cursor whether it is open or closed.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

See Also

Commands	close, declare cursor
Topics	Cursors

declare

Function

Declares the name and type of local variables for a batch or procedure. Local variables are assigned values with a **select** statement.

Syntax

Variable declaration:

```
declare @variable_name datatype
   [, @variable_name datatype]...
```

Variable assignment:

```
select @variable = {expression | select_statement}
   [, @variable = {expression | select_statement} ...]
   [from table_list]
   [where search_conditions]
   [group by group_by_list]
   [having search_conditions]
   [order by order_by_list]
   [compute function_list [by by_list]]
```

Keywords and Options

@variable_name – must begin with @ and must conform to the rules for identifiers.

datatype – can be either a system datatype or a user-defined datatype.

Examples

1.
```
declare @one varchar(18), @two varchar(18)
select @one = "this is one", @two = "this is two"
if @one = "this is one"
  print "you got one"
if @two = "this is two"
  print "you got two"
else print "nope"
```

2.
```
declare @veryhigh money
select @veryhigh = max(price)
  from titles
if @veryhigh > $20
  print "Ouch!"
```

Comments

- The maximum number of parameters in a procedure is 255. The number of local or global variables is limited only by available memory. The "@" sign denotes a variable name.
- Local variables are often used as counters for **while** loops or **if ...else** blocks. In stored procedures, they are declared for automatic, non-interactive use by the procedure when it executes. Local variables must be used within the batch or procedure in which they are declared.
- The **select** statement that assigns a value to the local variable usually returns a single value. If there is more than one value to return, the variable is assigned the last one. The **select** statement that assigns values to variables cannot be used to retrieve data in the same statement.
- The **print** and **raiserror** commands can take local variables as arguments.
- Users cannot create global variables, and cannot update the value of global variables directly in a select statement.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

See Also

Commands	**print, raiserror, select, while**
Topics	Parameters, Variables (Local and Global)

declare cursor

Function

Defines a cursor.

Syntax

```
declare cursor_name cursor
   for select_statement
   [for {read only | update [of column_name_list]}]
```

Parameters

cursor_name – is the name of the cursor being defined.

select_statement – is the query that defines the cursor result set. See **select** for more information.

for read only – specifies that the cursor result set cannot be updated.

for update – specifies that the cursor result set is updatable.

of *column_name_list* – is the list of columns from the cursor result set (specified by the *select_statement*) defined as updatable. SQL Server also allows you to include columns that are not specified in the list of columns of the cursor's *select_statement* (and excluded from the result set), but that are part of the tables specified in the *select_statement*.

Examples

1.
```
declare authors_crsr cursor
for select au_id, au_lname, au_fname
from authors
where state != 'CA'
```

   Defines a result set for the *authors_crsr* cursor that contains all authors from the *authors* table who do not reside in California.

2.
```
declare titles_crsr cursor
for select title, title_id from titles
where title_id like "BU%"
for read only
```

   Defines a read-only result set for the *titles_crsr* cursor that contains the business-type books from the *titles* table.

```
3. declare pubs_crsr cursor
   for select pub_name, city, state
   from publishers
   for update of city, state
```

Defines an updatable result set for the *pubs_crsr* cursor that contains all of the rows from the *publishers* table. It defines the address of each publisher (*city* and *state* columns) for update.

Comments

Restrictions on Cursors

- A **declare cursor** statement must precede any **open** statement for that cursor.
- You cannot include other statements with **declare cursor** in the same Transact-SQL batch.
- *cursor_name* must be a valid SQL Server identifier name (not longer than 30 characters and must start with a letter or the symbols # or _).

select *Statement*

- *select_statement* may use the full syntax and semantics of a Transact-SQL **select** statement, with these restrictions:
  - *select_statement* must contain a **from** clause.
  - *select_statement* cannot contain a **compute**, **for browse**, or **into** clause.
  - *select_statement* can contain the **holdlock** keyword.
- The *select_statement* can contain references to Transact-SQL parameter names or Transact-SQL local variables (for all cursor types except language– see "Cursors" in Chapter 3, ''Transact-SQL Topics'' for information about cursor types). The names must reference Transact-SQL parameters and local variables defined in the procedure, trigger, or statement batch that contains the **declare cursor** statement.

  The parameters and / or local variables referenced in the **declare cursor** statement do not have to contain valid values until the cursor is opened.
- The *select_statement* can contain references to the *inserted* and *deleted* temporary tables that are used in triggers.

Scope

- A cursor is defined by its **scope**. Scope determines the region where the cursor is known. Once a cursor's scope no longer exists, its cursor name also no longer exists. SQL Server divides the scope of a cursor into the following regions:
  - session – this region starts when a client logs onto SQL Server and ends when it logs off. This region precludes any regions defined by stored procedures or triggers.
  - stored procedure – this region starts when a stored procedure begins execution and ends when it completes execution. If a stored procedure calls another stored procedure, SQL Server starts a new region and treats it as a subregion of the first procedure.
  - trigger – this region starts when a stored procedure begins execution and ends when it completes execution.
- A cursor name must be unique within a given scope. Scopes are distinct in that a cursor name defined in one region can also be defined in another or within its own subregion. However, you cannot access a cursor defined in one region from another region. SQL Server does allow access to a cursor in subregions if no other cursor with that same name exists in the subregion.

  Name conflicts within a particular scope are detected by SQL Server only during runtime. For example, the following stored procedure works correctly because only one *names_crsr* cursor is defined in its scope:

```
create procedure proc1 as
    declare @flag int
    if (@flag)
        declare names_crsr cursor
        for select au_fname from authors
    else
        declare names_crsr cursor
        for select au_lname from authors
```

Result Set

- Cursor result set rows may not reflect the values in the actual base table rows. For example, a cursor declared with an **order by** clause usually requires the creation of an internal table to order the rows for the cursor result set. SQL Server does not lock the rows in the base table that correspond to the rows in the internal table, which permits other clients to update these base table rows. In that case, the rows returned to the client from the cursor result set would not be in sync with the base table rows.
- A cursor result set is generated as the rows are returned through a **fetch** of that cursor. This means that a cursor **select** query is processed like a normal **select** query. This process, known as a **cursor scan**, provides a faster turnaround time and eliminates the need to read rows the application does not require.

  A restriction of cursor scans is that they can only use the unique indexes of a table. However, if none of the base tables referenced by the cursor result set are updated by another process in the same lock space as that cursor, the restriction is unnecessary. SQL Server allows the declaration of cursors on tables without unique indexes, but any attempt to update these tables in the same lock space will close all cursors on such tables.

Updatability

- After defining a cursor using **declare cursor**, SQL Server determines if it is updatable or read-only. Updatable cursors implies that you can update or delete rows within the cursor result set. You cannot change the result set of read-only cursors.
- Use the **for update** or **for read only** clauses to explicitly define the cursor as updatable or read-only. However, you cannot define an updatable cursor if its *select_statement* contains one of the following constructs:

  - **distinct** option
  - **group by** clause
  - Aggregate functions
  - **union** operator

  If you omit either the **for update** or **read only** clause, SQL Server checks whether the cursor is updatable.

SQL Server also defines a cursor as read-only if you declare a Language or Server type cursor that includes an **order by** clause as part of its *select_statement*. SQL Server handles updates differently for Client or Executable type cursors, thereby eliminating this restriction.

- If you do not specify a *column_name_list* with the **for update** clause, all the specified columns in the query are updatable. SQL Server attempts to use unique indexes for updatable cursors when scanning the base table. For cursors, SQL Server considers an index containing an IDENTITY column to be unique, even if it is not declared so.

  If you do not specify the **for update** clause, SQL Server chooses any unique index, although it can also use other indexes or table scans if no unique index exists for the specified table columns. However, when you specify the **for update** clause, SQL Server must use a unique index defined for one or more of the columns to scan the base table. If none exists, it returns an error.

- Any columns of the base table you specify in the *column_name_list* of **for update** should include only those columns you need to update, and not any columns included in at least one unique index. This allows SQL Server to use that unique index for its cursor scan which helps prevent an update anomaly called the **Halloween Problem**.

  This problem occurs when a client updates a column of a cursor result set row which defines the order in which the rows are returned from the base tables. For example, if SQL Server accesses a base table using an index and the index key is updated by the client, the updated index row can move within the index and be read again by the cursor. This is a result of an updatable cursor only logically creating a cursor result set. The cursor result set is actually the base tables that derive the cursor.

- If you specify the **read only** option, the cursor result set may not be updated using the **delete** or **update** statement.

Standards and Compliance

Standard	Compliance Level	Comments
SQL89	Compliant	The **for update** and **for read only** options are Transact-SQL extensions.
SQL92	Entry level compliant	

See Also

Commands	**open**
Topics	Cursors

delete

Function

Removes rows from a table.

Syntax

```
delete [from] [[database.]owner.]{table_name|view_name}
   [where search_conditions]

delete [[database.]owner.]{table_name | view_name}
   [from [[database.]owner.]{table_name | view_name}
    [, [[database.]owner.]{table_name | view_name}]...]
   [where search_conditions]

delete [from] [[database.]owner.]{table_name|view_name}
   where current of cursor_name
```

Keywords and Options

from – (after **delete**) is an optional keyword used for compatibility with other versions of SQL. Follow it with the name of the table or view from which you want to remove rows.

from – (after *table_name* or *view_name*) lets you name more than one table or view to use with a **where** clause when specifying which rows to delete. This **from** clause allows you to delete rows from one table based on data stored in other tables, giving you much of the power of an embedded **select** statement.

where – is a standard **where** clause. See "**where** Clause" for more information.

where current of *cursor_name* – causes SQL Server to delete the row of the table or view indicated by the current cursor position for *cursor_name*.

Examples

1.
```
delete authors
```

   Deletes all rows from the *authors* table.

2.
```
delete from authors
where au_lname = "McBadden"
```

   Deletes a row or rows from the *authors* table.

3.
```
delete titles
from titles, authors, titleauthor
where authors.au_lname = 'Bennet'
  and authors.au_id = titleauthor.au_id
  and titleauthor.title_id = titles.title_id
```

   Deletes rows for books written by Bennet from the *titles* table. (The *pubs2* database includes a trigger (*deltitle*) which prevents the deletion of titles which are recorded in the *sales* table; you must drop this trigger for this example to work.)

4.
```
delete titles where current of title_crsr
```

   Deletes a row from the *titles* table currently indicated by the cursor *title_crsr.*

5.
```
delete authors
where syb_identity = 4
```

   Determines which row has a value of 4 for the IDENTITY column and deletes it from the *authors* table. Note the use of the **syb_identity** keyword instead of the actual name of the IDENTITY column.

Comments

Restrictions

- You cannot use **delete** with a multi-table view (one whose **from** clause names more than one table) even though you may be able to use **update** or **insert** on that same view. When you delete a row through a view you change multiple tables, which is not permitted. **insert** and **update** statements that affect only one base table of the view are permitted.
- SQL Server treats two different designations for the same table in a **delete** as two tables. For example, the following **delete** issued in *pubs2* specifies *discounts* as two tables (*discounts* and *pubs2..discounts)*:

  ```
  delete discounts
  from pubs2..discounts, pubs2..stores
  where pubs2..discounts.stor_id =
      pubs2..stores.stor_id
  ```

  In this case, the join does not include *discounts*, so the **where** condition remains true for every row; SQL Server deletes all rows in *discounts* (which is not the desired result). To avoid this problem, use the same designation for a table throughout the statement.

Deleting All Rows from a Table

- If you don't use a **where** clause, **all** rows in the table named after **delete** [**from**] are removed. The table, though empty of data, continues to exist until you issue a **drop table** command.
- **truncate table** and **delete** without a row specification are functionally equivalent, but **truncate table** is faster. **delete** removes rows one at a time and logs these transactions; **truncate table** removes whole data pages, and the rows are not logged.

  Both **delete** and **truncate table** reclaim the space occupied by the data and its associated indexes.

delete *and Transactions*

- When you set chained transaction mode, SQL Server implicitly begins a transaction with the **delete** statement if no transaction is currently active. To complete any deletes, you must **commit** the transaction, or you can **rollback** the changes. For example:

```
delete from sales where date < '01/01/89'
if exists (select stor_id
    from stores
    where stor_id not in
    (select stor_id from sales))
        rollback transaction
else
        commit transaction
```

  This batch begins a transaction (using the chained transaction mode) and deletes from the *sales* table those rows with dates earlier than Jan. 1, 1989. If it deletes all sales entries associated with a store, it rolls back all the changes to *sales* and ends the transaction. Otherwise, it commits the deletions and ends the transaction. For more information about the chained mode, see "Transactions".

Delete Triggers

- You can define a trigger that will take a specified action when a **delete** command is issued on a specified table.

Using **delete where current of**

- Use the clause **where current of** with cursors. Before deleting rows using the clause **where current of**, you must first define the cursor with **declare cursor** and open it using the **open** statement. Position

the cursor on the row you want to delete using one or more **fetch** statements. The cursor name cannot be a Transact-SQL parameter or local variable. The cursor must be an updatable cursor or SQL Server returns an error. Any deletion to the cursor result set also affects the base table row which the cursor row is derived from. You can delete only one row at a time using the cursor.

- You cannot delete rows in a cursor result set if the cursor's **select** statement contains a join clause even though the cursor is considered updatable. The *table_name* or *view_name* specified with a **delete...where current of** must be the table or view specified in the first **from** clause of the **select** statement that defines the cursor.
- After the deletion of a row from the cursor's result set, the position of the cursor points before the next row in the cursor's result set. You must still use **fetch** to access the next row. If the deleted row is the last row of the cursor result set, the cursor points after the last row of the result set. The following describes the position and behavior of open cursors affected by a **delete**:
  - If a client deletes a row (using another cursor or a regular **delete**) and that row represents the current cursor position of other opened cursors owned by the same client, the position of each affected cursor is implicitly set to before the next available row. However, it is not possible for one client to delete a row representing the current cursor position of another client's cursor.
  - If a client deletes a row which represents the current cursor position of another cursor defined by a join operation and owned by the same client, SQL Server still accepts the **delete** statement. However, it implicitly closes the cursor defined by the join.

Standards and Compliance

Standard	Compliance Level	Comments
SQL89	Compliant	The use of more than one table in the **from** clause and qualification of table name with database name are Transact-SQL extensions.
SQL92	Entry level compliant	

Permissions

delete permission defaults to the table or view owner, who can transfer it to other users.

If you have **set ansi_permissions on**, you must have **select** permission on all columns appearing in the **where** clause, in addition to the regular permissions required for **delete** statements. (By default, **ansi_permissions** is **off**.)

See Also

Commands	**create trigger, drop table, drop trigger, truncate table, where Clause**
Topics	Cursors, Transactions

disk init

Function

Makes a physical device or file usable by SQL Server. (The master device is initialized by the **sybinit** installation program; it is not necessary to initialize this device with **disk init**.)

Syntax

```
disk init
   name = "device_name" ,
   physname = "physicalname" ,
   vdevno = virtual_device_number ,
   size = number_of_blocks
   [, vstart = virtual_address ,
   cntrltype = controller_number ]
   [, contiguous] (OpenVMS only)
```

Keywords and Options

name – is the name of the database device or file. The name must conform to the rules for identifiers, and be enclosed in single or double quotes. This name is used in the **create database** and **alter database** commands.

physname – is the full specification of the database device. This name must be enclosed in single or double quotes.

vdevno – is the virtual device number. It must be unique among database devices associated with SQL Server. The device number 0 is reserved for the master device. Legal numbers are between 1 and 255, but the highest number must be one less than the number of database devices for which your SQL Server is configured. For example, for a SQL Server with the default configuration of 10 devices, the legal device numbers are 1 through 9. To see the maximum number of devices available on your SQL Server, run **sp_configure**, and check the **devices** value.

To determine the virtual device number, look at the *device_number* column of the **sp_helpdevice** report, and use the next unused integer.

If you drop a device with **sp_dropdevice**, you cannot re-use its **vdevno** until the server is re-booted.

size – is the size of the database device in 2K blocks.

If you are planning to use the new device for the creation of a new database, the minimum **size** is the size of the *model* database, 1024 2K blocks (2 MB). If you are initializing a log device, the **size** can be as small as 512 2K blocks (1 MB).

vstart – is the starting virtual address, or the starting offset in 2K blocks. The value for **vstart** should be 0 (the default). Reset **vstart** only if instructed to do so.

cntrltype – specifies the disk controller. Its default value is zero. Reset **cntrltype** only if instructed to do so.

contiguous – forces contiguous database file creation (*OpenVMS only*). This option is meaningful only if the mirror is a file rather than a foreign device. If you include the **contiguous** option, the system creates a contiguous file or the command fails with an error message. If you do not include the **contiguous** option, the system still tries to create a contiguous file. If it fails to create the file contiguously, the system creates a file that does not force contiguity. In either case, the system prints a message indicating the type of file that is created.

Examples

1.
```
disk init
  name = "user_disk",
  physname = "/dev/rxy1a",
  vdevno = 2, size = 5120
```

   Initializes a disk on a UNIX system.

2.
```
disk init
 name = “user_disk”,
 physname = “disk$rose_1:[dbs]user.dbs”,
 vdevno = 2, size = 5120,
 contiguous
```

   Initializes a disk on a OpenVMS system, forcing the database file to be created contiguously.

Comments

- Use **disk init** for each new database device. Each time **disk init** is issued, a row is added to *master..sysdevices*. A new database device does not automatically become part of the pool of default database storage. Assign default status to a database device with the system procedure **sp_diskdefault**.

- On OpenVMS systems, using a logical name to refer to the **physname** offers more flexibility than using a hard-coded path name. For example, if you define the logical name "userdisk" as:

  ```
  disk$rose_1:[dbs]user.dbs
  ```

  you can change the **physname** in the OpenVMS example above to "*userdisk*". If you decide to reorganize your disk or to move "*user.dbs*", all you need to do is redefine the logical name as the new path.

  Any logical name used by a particular SQL Server must be:

  - A system logical name, or
  - A process logical name defined in the runserver file for that SQL Server.

- Back up the *master* database with the **dump database** or **dump transaction** command after each use of **disk init.** This makes recovery easier and safer in case *master* is damaged. (If you add a device with **disk init** and fail to back up *master*, you may be able to recover the changes by using **disk reinit** and then stopping and restarting SQL Server.)
- User databases are assigned to database devices with the optional clause:

  on ***device_name***

  of the **create database** or **alter database** command.
- The preferred method for placing a database's transaction log (that is, the system table *syslogs*) on a different device than the one on which the rest of the database is stored, is the **log on** extension to **create database.** Alternatively, you can name at least two devices when you create the database, and then execute the system procedure **sp_logdevice.** You can also **alter database** to a second device and then run **sp_logdevice.**

➤ ***Note***

The **log on** extension immediately moves the entire log to a separate device; the **sp_logdevice** method retains part of the system log on the original database device, until transaction activity causes the migration to become complete.

- For a report on all SQL Server devices on your system (both database and dump devices), execute the system procedure **sp_helpdevice.**

- Remove a database device with the system procedure **sp_dropdevice**. You must first drop all existing databases on that device.

  After dropping a database device, you can create a new one with the same name (using **disk init**) as long as you give it a different physical name and virtual device number. If you wish to use the same physical name and virtual device number, you must restart SQL Server.
- If **disk init** failed because you gave a **size** that was too large for the database device, you should use a different virtual device number, or restart the Server before executing **disk init** again.
- To successfully complete disk initialization, the "sybase" user must have the appropriate operating system permissions on the device that is being initialized.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

Permissions

disk init permission defaults to System Administrators, and is not transferable. You must be using the *master* database to use **disk init**.

See Also

Commands	**alter database, create database, disk refit, disk reinit, dump database, dump transaction, load database, load transaction**
System procedures	**sp_diskdefault, sp_dropdevice, sp_helpdevice, sp_logdevice**

disk mirror

Function

Creates a software mirror that immediately takes over when the primary device fails. You can mirror the master device, devices that store data, and devices that store transaction logs; you cannot mirror dump devices.

Syntax

```
disk mirror
   name = "device_name" ,
   mirror = "physicalname"
   [ ,writes = { serial | noserial }]
   [ ,contiguous ] (OpenVMS only)
```

Keywords and Options

name – is the name of the database device that you want to mirror. This is recorded in the *name* column of the *sysdevices* table. The name must be enclosed in single or double quotes.

mirror – is the full path name of the database mirror device that is to be your secondary device. It must be enclosed in single or double quotes. If the secondary device is a file, *physicalname* should be a path specification sufficient to identify the file unambiguously. It cannot be an existing file.

writes – allows you to choose whether to enforce serial writes to the devices. In the default case (**serial**), the write to the primary database device is guaranteed to finish before the write to the secondary device begins. If the primary and secondary devices are on different physical devices, serial writes can ensure that at least one of the disks may be unaffected in the event of a power failure.

contiguous – (*OpenVMS only*) is meaningful only if the mirror is a file rather than a foreign device. This option forces the file that will be used as the secondary device to be created contiguously. If you include the **contiguous** option, the system creates a contiguous file or the command fails with an error message. If you do not include the **contiguous** option, the system still tries to create a contiguous file. If it fails to create the file contiguously, the system creates a file that does not force contiguity. In either case, the system prints

a message indicating the type of file that is created. The **contiguous** option is also available with **disk init** for OpenVMS users.

Examples

1.
```
disk mirror
  name = "user_disk",
  mirror = "/server/data/mirror.dat"
```

   Creates a software mirror for the database device *user_disk* on the file *mirror.dat*.

Comments

- Devices are mirrored, not databases.
- A device and its mirror together constitute one logical device. The physical name of the mirror device goes in the *mirrorname* column of the *sysdevices* table; it does not require a separate entry in *sysdevices* and should not be initialized with **disk init.**
- Mirror all default database devices so that you are still protected if a **create** or **alter database** command affects a database device in the default list.
- You can mirror or unmirror database devices without shutting down SQL Server. Disk mirroring does not interfere with ongoing activities in the database.
- Back up the *master* database with the **dump database** command after each use of **disk mirror.** This makes recovery easier and safer in case *master* is damaged.
- When a read or write to a mirrored device is unsuccessful, SQL Server causes the bad device to become unmirrored, and prints error messages. SQL Server continues to run, unmirrored. The System Administrator must use the **disk remirror** command to restart mirroring.
- Always put user database transaction logs on a separate database device. To put a database's transaction log (that is, the system table *syslogs*) on a different device than the one on which the rest of the database is stored, name the database device and the log device when you create the database. You could also **alter database** to a second device and then run the system procedure **sp_logdevice.**
- For greater protection, mirror the database device used for transaction logs.
- If you mirror the database device for the *master* database, you can use the **-r** option and the name of the mirror for UNIX, or the

mastermirror option for OpenVMS, when you restart SQL Server with the **dataserver** utility program. This option should also be added to the *RUN_servername* file for that server, so that the **startserver** utility program will know about it. For example:

```
dataserver -dmaster.dat -rmirror.dat
```

starts a master device named *master.dat* and its mirror, *mirror.dat*. For more information, see **dataserver** and **startserver** in the *SQL Server Utility Programs*.

- If you mirror a database device that has unallocated space (room for additional **create database** and **alter database** statement to allocate part of the device), **disk mirror** begins mirroring these allocations when they are made, not when the **disk mirror** command is issued.
- For a report on all SQL Server devices on your system (user database devices and their mirrors, as well as dump devices), execute the system procedure **sp_helpdevice**.
- For more details about disk mirroring in the OpenVMS environment, see the *SYBASE SQL Server Installation Guide for OpenVMS*.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

Permissions

disk mirror permission defaults to the System Administrator, and is not transferable. You must be using the *master* database to use **disk mirror**.

See Also

Commands	**alter database, create database, disk init, disk refit, disk reinit, disk remirror, disk unmirror, dump database, dump transaction, load database, load transaction**
Topics	Disk Mirroring
System procedures	**sp_diskdefault, sp_helpdevice, sp_logdevice**
Utility programs	**dataserver, startserver**

disk refit

Function

Rebuilds the *master* database's *sysusages* and *sysdatabases* system tables from information contained in *sysdevices*. Use **disk refit** after **disk reinit** as part of the procedure to restore the *master* database.

Syntax

```
disk refit
```

Examples

```
disk refit
```

Comments

- Stop and then restart SQL Server after using **disk refit.**
- For more information, see the *SQL Server Installation Guide* for your system or the *System Administration Guide.*

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

Permissions

disk refit permission defaults to System Administrators, and is not transferable. You must be in the *master* database to use **disk refit.**

See Also

Commands	**disk init, disk reinit**
System procedures	**sp_addumpdevice, sp_helpdevice**

disk reinit

Function

Rebuilds the *master* database's *sysdevices* system table. Use **disk reinit** as part of the procedure to restore the *master* database.

Syntax

```
disk reinit
   name = "device_name",
   physname = "physicalname" ,
   vdevno = virtual_device_number ,
   size = number_of_blocks
   [, vstart = virtual_address ,
   cntrltype = controller_number]
```

Keywords and Options

name – is the name of the database device. It must conform to the rules for identifiers, and must be enclosed in single or double quotes. This name is used in the **create database** and **alter database** commands.

physname – is the name of the database device. The physical name must be enclosed in single or double quotes.

vdevno – is the virtual device number. It must be unique among devices used by SQL Server. The device number 0 is reserved for the *master* database device. Legal numbers are between 1 and 255, but cannot be greater than the number of database devices for which your system is configured. The default is 50 devices.

size – is the size of the database device in 2K-blocks. The minimum usable size is 1024 2K-blocks (2 MBs).

vstart – is the starting virtual address, or the starting offset in 2K blocks. The value for vstart should be 0 (the default). Reset it only if instructed to do so.

cntrltype – specifies the disk controller. Its default value is 0. Reset it only if instructed to do so.

Examples

```
disk reinit
  name = "user_disk",
  physname = "/server/data/userdata.dat",
  vdevno = 2, size = 5120
```

Comments

- **disk reinit** insures that *master..sysdevices* is correct if the *master* database has been damaged, or if devices have been added since the last dump of *master*.
- **disk reinit** is very similar to **disk init**, but does not initialize the database device.
- For complete information on restoring the *master* database, see the SYBASE *SQL Server Installation Guide* for your system or the *System Administration Guide*.

Standards and Compliance

Standard	Compliance level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

Permissions

disk reinit permission defaults to System Administrators, and is not transferable. You must be in the *master* database to use **disk reinit**.

See Also

Commands	**alter database, create database, dbcc, disk init, disk refit**
System procedures	**sp_addumpdevice, sp_helpdevice**

disk remirror

Function

Restarts disk mirroring after it is stopped by failure of a mirrored device or temporarily disabled by the **disk unmirror** command.

Syntax

```
disk remirror
   name = "device_name"
```

Keywords and Options

name – is the name of the database device that you want to remirror. This is recorded in the *name* column of the *sysdevices* table. The name must be enclosed in single or double quotes.

Examples

```
1. disk remirror
     name = "user_disk"
```

Resumes software mirroring on the database device *user_disk*.

Comments

- Devices are mirrored, not databases.
- You should mirror all the default database devices so that you are still protected if a **create** or **alter database** command affects a database device in the default list.
- You can mirror, remirror, or unmirror database devices without shutting down SQL Server. Disk mirroring does not interfere with ongoing activities in the database.
- It is important to back up the *master* database with the **dump database** command after each use of **disk remirror**. This makes recovery easier and safer in case *master* is damaged.
- When a read or write to a mirrored device is unsuccessful, SQL Server causes the bad device to become unmirrored, and prints error messages. SQL Server continues to run, unmirrored. The System Administrator must use **disk remirror** to restart mirroring.
- Use **disk remirror** to reestablish mirroring after it is temporarily stopped with the **mode = retain** option of the **disk unmirror** command. The **disk remirror** command copies data on the retained disk to the mirror. If mirroring is permanently disabled with the **mode =**

remove option, you must remove the operating system file that contains the mirror before using **disk remirror.**

- In addition to mirroring user database devices, you should always put user database transaction logs on a separate database device. The database device used for transaction logs can also be mirrored, for even greater protection. To put a database's transaction log (that is, the system table *syslogs*) on a different device than the one on which the rest of the database is stored, name the database device and the log device when you create the database. You could also **alter database** to a second device and then run the system procedure **sp_logdevice.**
- If you mirror the database device for the *master* database, you can use the **-r** option and the name of the mirror for UNIX, or the **mastermirror** option for OpenVMS, when you restart SQL Server with the **dataserver** utility program. This option should also be added to the *RUN_servername* file for that server, so that the **startserver** utility program will know about it. For example:

  ```
  dataserver -dmaster.dat -rmirror.dat
  ```

  starts a master device named *master.dat* and its mirror, *mirror.dat*. For more information, see **dataserver** and **startserver** in the *SQL Server Utility Programs*.
- For a report on all SQL Server devices on your system (user database devices and their mirrors, as well as dump devices), execute the system procedure **sp_helpdevice.**
- For more details about disk mirroring in the OpenVMS environment, see the *SYBASE SQL Server Installation Guide for OpenVMS*.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

Permissions

disk remirror permission defaults to the System Administrator, and is not transferable. You must be using the *master* database to use **disk remirror.**

See Also

Commands	**alter database, create database, disk init, disk mirror, disk refit, disk reinit, disk unmirror, dump database, dump transaction, load database, load transaction**
Topics	Disk Mirroring
System procedures	**sp_diskdefault, sp_helpdevice, sp_logdevice**
Utility programs	**dataserver, startserver**

disk unmirror

Function

De-activates disk mirroring to allow hardware maintenance or the changing of a hardware device. **disk unmirror** disables either the original database device or the mirror, so that it is no longer available to SQL Server for reads or writes. It does not remove the associated file from the operating system.

Syntax

```
disk unmirror
   name = "device_name"
   [ ,side = { "primary" | secondary }]
   [ ,mode = { retain | remove }]
```

Keywords and Options

name = *"device_name"* – is the name of the database device that you want to unmirror. The name must be enclosed in single or double quotes.

side – specifies whether to disable the "**primary**" device or the **secondary** device (the mirror). By default, the secondary device is unmirrored.

mode – determines whether the unmirroring is temporary (**retain**) or permanent (**remove**). By default, unmirroring is temporary.

Set **retain** when you plan to remirror the database device later in the same configuration. This option mimics what happens when the primary device fails:

- I/O is directed only at the device *not* being unmirrored
- The *status* column of *sysdevices* indicates that mirroring is deactivated

remove eliminates all *sysdevices* references to a mirror device:

- The *status* column indicates that the mirroring feature is ignored
- The *phyname* column is replaced by the name of the secondary device in the *mirrorname* column if the primary device is the one being deactivated
- The *mirrorname* column is set to NULL

Examples

1. **disk unmirror**
   name = "user_disk"

   Suspends software mirroring for the database device *user_disk*.

Comments

- You can unmirror a database device while it is in use.
- You cannot unmirror any of a database's devices while a **dump database**, **load database**, or **load transaction** command is in progress for that database. SQL Server displays a message asking you whether to abort the dump or load or to defer the **disk unmirror** until after the dump or load completes.
- You cannot unmirror a database's log device while a **dump transaction** is in progress for that database, SQL Server displays a message asking you whether to abort the dump or defer the **disk unmirror** until after the dump completes.

➤ ***Note***

dump transaction with truncate_only and **dump transaction with no_log** are not affected when a log device is unmirrored.

- Disk unmirroring alters the *sysdevices* table in the *master* database. It is important to back up the *master* database with the **dump database** command after each use of **disk unmirror.** This makes recovery easier and safer in case *master* is damaged.
- Use **disk remirror** to reestablish mirroring after it is temporarily stopped with the **mode = retain** option of the **disk unmirror** command. If mirroring is permanently disabled with the **mode = remove** option, you must remove the operating system file that contains the mirror before using **disk remirror.**
- You should mirror all the default database devices so that you are still protected if a **create** or **alter database** command affects a database device in the default list.
- When a read or write to a mirrored device is unsuccessful, SQL Server automatically unmirrors the bad device and prints error messages. SQL Server continues to run, unmirrored. A System Administrator must restart mirroring with the **disk remirror** command.

- For a report on all SQL Server devices on your system (user database devices and their mirrors, as well as dump devices), execute the system procedure **sp_helpdevice.**
- For more details about disk mirroring in the OpenVMS environment, see the *SYBASE SQL Server Installation Guide for OpenVMS*.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

Permissions

disk unmirror permission defaults to the System Administrator, and is not transferable. You must be using the *master* database to use **disk unmirror.**

See Also

Commands	**alter database**, **create database**, **disk init**, **disk mirror**, **disk refit**, **disk reinit**, **disk remirror**, **dump database**, **dump transaction**, **load database**, **load transaction**
Topics	Disk Mirroring
System procedures	**sp_diskdefault, sp_helpdevice, sp_logdevice**
Utility programs	**dataserver, startserver**

drop database

Function

Removes one or more databases from SQL Server.

Syntax

drop database *database_name* [, *database_name*]...

Keywords and Options

database_name – is the name of a database to remove. Use **sp_helpdb** to get a list of databases.

Examples

```
1. drop database publishing
2. drop database publishing, newpubs
```

The dropped databases (and their contents) are gone.

Comments

- You must be using the *master* database to drop a database.
- Removing a database deletes the database and all its objects, frees its storage allocation, and erases its entries in the *sysdatabases* and *sysusages* system tables in the *master* database.
- You cannot drop a database that is in use (open for reading or writing by any user).
- You cannot use **drop database** to remove a database that is referenced by a table in another database. Execute the following query to determine which tables and external databases have foreign key constraints on primary key tables in the current database:

  ```
  select object_name(tableid), db_name(frgndbname)
  from sysreferences
  where frgndbname is not null
  ```

  Use **alter table** to drop these cross-database constraints, then re-issue the **drop database** command.
- You cannot use **drop database** to remove a damaged database. Use the **dbcc dbrepair** command:

  dbcc dbrepair (*database_name*, dropdb)

- You cannot drop the *sybsecurity* database if auditing is enabled. If auditing is disabled, only the System Security Officer can drop *sybsecurity*.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

Permissions

Only the Database Owner can execute **drop database**, except in the case of the *sybsecurity* database which can only be dropped by the System Security Officer.

See Also

Commands	**alter database, create database, dbcc, use**
System procedures	**sp_changedbowner, sp_helpdb, sp_renamedb, sp_spaceused**

drop default

Function

Removes a user-defined default.

Syntax

```
drop default [owner.]default_name
   [, [owner.]default_name]...
```

Keywords and Options

default_name – is the name of an existing default. Execute **sp_help** to get a list of existing defaults.

Examples

```
drop default datedefault
```

The user-defined default *datedefault* has been dropped.

Comments

- A default cannot be dropped if it is currently bound to a column or user-defined datatype. Use the system procedure **sp_unbindefault** to unbind the default before you drop it.
- You can bind a new default to a column or user-defined datatype without unbinding its current default. The new default will override the old one.
- When you drop a default for a NULL column, NULL becomes the column's default value. When you drop a default for a NOT NULL column, an error message will display if users do not explicitly enter a value for that column.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

Permissions

drop default permission defaults to the owner of the default, and is not transferable.

See Also

Commands	**create default**
System procedures	**sp_help, sp_helptext, sp_unbindefault**

drop index

Function

Removes an index from a table in the current database.

Syntax

```
drop index table_name.index_name
   [, table_name.index_name]...
```

Keywords and Options

table_name – is the table in which the indexed column is located. The table must be in the current database.

index_name – is the index to be dropped. In Transact-SQL, index names need not be unique in a database, though they must be unique within a table.

Examples

```
drop index authors.au_id_ind
```

The index *au_id_ind* in the *authors* table no longer exists.

Comments

- Once the command is issued, you regain all of the space occupied by the index. This space can be used for any database objects.
- You cannot use **drop index** on any of the system tables that exist in the *master* database and in user databases.
- You cannot drop indexes supporting unique constraints using the **drop index** statement. Such indexes are dropped when the constraints are dropped through **alter table** or when the table is dropped. See **create table** for more information about unique constraint indexes.
- You cannot drop indexes that are currently used by any open cursor. For information about which cursors are open and what indexes they use, use **sp_cursorinfo**.
- To get information about what indexes exist on a table or view, use:

  ```
  sp_helpindex tabview_name
  ```

  where *tabview_name* is the name of the table or view.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

Permissions

drop index permission defaults to the index owner, and is not transferable.

See Also

Commands	**create index**
System procedures	**sp_cursorinfo, sp_helpindex, sp_spaceused**

drop procedure

Function

Removes user-defined stored procedures.

Syntax

```
drop procedure [owner.]procedure_name
   [, [owner.]procedure_name] ...
```

Keywords and Options

procedure_name – is the name of the stored procedure to be dropped.

Examples

```
drop procedure showind
```

The stored procedure *showind* has been deleted.

Comments

- The existence of a stored procedure is checked each time a user or a program executes that procedure.
- A procedure group (more than one procedure with the same name but with different *;number* suffixes) can be dropped with a single **drop procedure** statement. For example, if the procedures used with the application *orders* were named *orderproc;1*, *orderproc;2*, etc., the following statement:

  ```
  drop proc orderproc
  ```

  would drop the entire group. Once procedures have been grouped, individual procedures within the group cannot be dropped. For example, the statement:

  ```
  drop procedure orderproc;2
  ```

  is not allowed.
- The system procedure **sp_helptext** displays the procedure's text (which is stored in *syscomments*).
- **drop procedure** drops user-created procedures only from your current database.

Standards and Compliance

Standard	Compliance level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

Permissions

drop procedure permission defaults to the procedure owner, and is not transferable.

See Also

Commands	**create procedure**
System procedures	**sp_depends, sp_helptext, sp_rename**

drop rule

Function

Removes a user-defined rule.

Syntax

```
drop rule [owner.]rule_name [, [owner.]rule_name]...
```

Examples

```
drop rule pubid_rule
```

The rule *pubid_rule* no longer exists.

Keywords and Options

rule_name – is the name of the rule to be dropped.

Comments

- Before dropping a rule, you must unbind it using the system procedure **sp_unbindrule**. If the rule hasn't been unbound, an error message will be given and the **drop rule** command will be aborted.
- You can bind a new rule to a column or user-defined datatype without unbinding its current rule. The new rule overrides the old one.
- After you drop a rule, new data entered into the columns that previously were governed by it goes in without constraints. Existing data is not affected in any way.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

Permissions

drop rule permission defaults to the rule owner, and is not transferable.

See Also

Commands	**create rule**
System procedures	**sp_bindrule, sp_help, sp_helptext, sp_unbindrule**

drop table

Function

Removes a table definition and all of its data, indexes, triggers, and permission specifications from the database.

Syntax

```
drop table [[database.]owner.]table_name
   [, [[database.]owner.]table_name ]...
```

Keywords and Options

table_name – is the name of the table to be dropped.

Examples

```
drop table roysched
```

The table *roysched* and its data and indexes no longer exist.

Comments

- When you **drop** a table, any rules or defaults on it lose their binding, and any triggers associated with it are automatically dropped. If you recreate a table, you must re-bind the appropriate rules and defaults and recreate any triggers.
- The system tables affected when a table is dropped are *sysobjects*, *syscolumns*, *sysindexes*, *sysprotects*, and *syscomments*.
- You cannot use the **drop table** command on any of the system tables, either in the *master* database or in a user database.
- You can drop a table in any database so long as you are the table owner. For example, to drop a table called *newtable* in the database *otherdb*:

  ```
  drop table otherdb..newtable
  ```

  or:

  ```
  drop table otherdb.yourname.newtable
  ```

- If you **delete** all the rows in a table or give the **truncate table** command, the table still exists until you **drop** it.

Using Cross-Database Referential Integrity Constraints

- When you create a cross-database constraint, SQL Server stores the following information in the *sysreferences* system table of each database:

Information Stored in *sysreferences*	Columns with Information about the	
	Referenced Table	Referencing Table
Key Column IDs	*refkey1* through *refkey16*	*fokey1* through *fokey16*
Table ID	*reftabid*	*tableid*
Database Name	*pmrydbname*	*frgndbname*

Table 1-11: Information stored about referential integrity constraints

- You can drop the referencing table or its database without problems. SQL Server automatically removes the foreign key information from the referenced database.
- Because the referencing table depends on information from the referenced table, SQL Server does not allow you to:
  - Drop the referenced table,
  - Drop the external database that contains it, or
  - Rename either database with **sp_renamedb**

  Use the **sp_helpconstraint** system procedure to determine which tables reference the table you want to drop. Use **alter table** to drop the constraints before re-issuing the **drop table** command.
- Each time you add or remove a cross-database constraint, or drop a table that contains a cross-database constraint, dump **both** of the affected databases.

◆ ***WARNING!***

Loading earlier dumps of these databases could cause database corruption.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

Permissions

drop table permission defaults to the table owner and is not transferable, except to the Database Owner, who can impersonate the table owner by running the **setuser** command.

See Also

Commands	alter table, create table, delete, truncate table
System procedures	sp_depends, sp_help, sp_spaceused

drop trigger

Function

Removes a trigger.

Syntax

```
drop trigger [owner.]trigger_name
   [, [owner.]trigger_name]...
```

Keywords and Options

trigger_name – is the name of the trigger to be dropped.

Examples

```
drop trigger trigger1
```

The trigger *trigger1* no longer exists.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

Permissions

drop trigger permission defaults to the trigger owner, and is not transferable.

See Also

Commands	create trigger
System procedures	sp_depends, sp_help, sp_helptext

drop view

Function

Removes one or more views from the current database.

Syntax

```
drop view [owner.]view_name [, [owner.]view_name]...
```

Keywords and Options

view_name – is the name of the view to be dropped. It must be a legal identifier. It cannot include a database name.

Examples

```
drop view new_price
```

Removes the view *new_price* from the current database.

Comments

- When you use **drop view**, the definition of the view and other information about it is deleted from the system tables *sysobjects*, *syscolumns*, *syscomments*, *sysdepends*, *sysprocedures*, and *sysprotects*. All privileges for the view are deleted, too.
- Existence of a view is checked each time the view is referenced, for example, by another view or by a stored procedure.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

Permissions

drop view permission defaults to the view owner, and is not transferable.

See Also

Commands	**create view**
System procedures	**sp_depends, sp_help, sp_helptext**

dump database

Function

Makes a backup copy of the entire database, including the transaction log, in a form that can be read in with **load database**. Dumps and loads are performed through a Backup Server.

Syntax

```
dump database database_name
   to stripe_device [ at backup_server_name ]
       [density = density_value,
        blocksize = number_bytes,
        capacity = number_kilobytes,
        dumpvolume = volume_name,
        file = file_name]
   [stripe on stripe_device [ at backup_server_name ]
       [density = density_value,
        blocksize = number_bytes,
        capacity = number_kilobytes,
        dumpvolume = volume_name,
        file = file_name]]
   [[stripe on stripe_device [ at backup_server_name ]
       [density = density_value,
        blocksize = number_bytes,
        capacity = number_kilobytes,
        dumpvolume = volume_name,
        file = file_name]]...]
   [with {
        density = density_value,
        blocksize = number_bytes,
        capacity = number_kilobytes,
        dumpvolume = volume_name,
        file = file_name,
       [dismount | nodismount],
       [nounload | unload],
       retaindays = number_days,
       [noinit | init],
       notify = {client | operator_console}
       }]]
```

Keywords and Options

database_name – is the name of the database from which you are copying data. The database name can be specified as a literal, a local variable, or a parameter to a stored procedure.

to *stripe_device* – is the device to which data is being copied. See "Specifying Dump Devices" for information about what form to use when specifying a dump device. See the *System Administration Guide Supplement* for a list of supported dump devices.

at *backup_server_name* – is the name of the Backup Server. This parameter is optional when dumping to the default Backup Server. When dumping across the network, specify the *network name* of a remote Backup Server running on the machine to which the dump device is attached. For platforms that use *interfaces* files, the *backup_server_name* must appear in the *interfaces* file.

density = *density_value* – overrides the default density for a tape device. **Use this option only when reinitializing a volume on OpenVMS systems**. Valid densities are 800, 1600, 6250, 6666, 10000, and 38000. Not all values are valid for every tape drive; use the correct density for your tape drive.

blocksize = *number_bytes* – overrides the default blocksize for a dump device. **(Wherever possible, use the default blocksize**; it is the "best" blocksize for your system.) The blocksize must be at least one database page (2048 bytes for most systems, 4096 for Stratus) and must be an exact multiple of the database page size. On OpenVMS systems, blocksize cannot exceed 16,384 bytes.

capacity = *number_kilobytes* – is the maximum amount of data that the device can write to a single tape volume. The capacity must be at least five database pages, and should be slightly less than the recommended capacity for your device.

OpenVMS systems write until they reach the physical end-of-tape marker, then send a volume change request. On UNIX platforms that cannot reliably detect the end-of-tape marker, you must indicate how many kilobytes can be dumped to the tape. You **must** supply a **capacity** for dump devices specified as a physical pathname. If a dump device is specified as a logical device name, the Backup Server uses the *size* parameter stored in the *sysdevices* system table unless you specify a capacity.

dumpvolume = *volume_name* – establishes the name that is assigned to the volume. The Backup Server writes the *volume_name* in the ANSI tape label when overwriting an existing dump, dumping to a brand new tape, or dumping to a tape whose contents are not recognizable. The **load database** command checks the label and generates an error message if the wrong volume is loaded.

◆ ***WARNING!***

Be sure to label each tape volume as you create it. This makes it easier for the operator to load the correct tape.

stripe on *stripe_device* – is an additional dump device. You can use up to 32 devices, including the device named in the **to** *stripe_device* clause. The Backup Server splits the database into approximately equal portions, and sends each portion to a different device. Dumps are made concurrently on all devices, reducing the time required to make a dump and requiring fewer volume changes during the dump. See "Specifying Dump Devices" for information about how to specify a dump device.

dismount | **nodismount** – **on platforms, such as OpenVMS, that support logical dismount**, determines whether tapes remain mounted. By default, all tapes used for a dump are dismounted when the dump completes. Use **nodismount** to keep tapes available for additional dumps or loads.

nounload | **unload** – determines whether tapes rewind after the dump completes. By default, tapes do not rewind, allowing you to make additional dumps to the same tape volume. Specify **unload** for the last dump file to be added to a multi-dump volume. This rewinds and unloads the tape when the dump completes.

retaindays = *number_days* – **on UNIX systems**, specifies the number of days that Backup Server protects you from overwriting a dump. **This option is meaningful for disk, quarter-inch cartridge, and single-file media. On multi-file media, this option is meaningful for all volumes but the first.** If you try to overwrite a dump before it expires, Backup Server requests confirmation before overwriting the unexpired volume.

The *number_days* must be a positive integer, or 0 for dumps you can overwrite immediately. If you do not specify a **retaindays** value, Backup Server uses the server-wide "tape retention" value, set by **sp_configure.**

noinit | **init** – determines whether the dump is appended to existing dump files or reinitializes (overwrites) the tape volume. By default, dumps are appended following the last end-of-tape mark, allowing you to dump additional databases to the same volume. New dumps can be appended only to the last volume of

a multi-volume dump. Use **init** for the first database you dump to a tape, to overwrite its contents.

file = *file_name* – is the name of the dump file. The name cannot exceed 17 characters and must conform to operating system conventions for file names. If you do not specify a file name, Backup Server creates a default. For more information, see ''Dump Files''.

notify = {**client** | **operator_console**} – overrides the default message destination.

- On operating systems (such as OpenVMS) that offer an operator terminal feature, volume change messages are always sent to the operator terminal on the machine on which the Backup Server is running. Use **client** to route other Backup Server messages to the terminal session that initiated the **dump database**.
- On operating systems (such as UNIX) that do not offer an operator terminal feature, messages are sent to the client that initiated the **dump database**. Use **operator_console** to route messages to the terminal on which the Backup Server is running.

Examples

1. For UNIX:

```
dump database pubs2
  to "/dev/nrmt0"
```

   For OpenVMS:

```
dump database pubs2
  to "MTA0:"
```

   Dumps the database *pubs2* to a tape device.If the tape has an ANSI tape label, this command appends this dump to the files already on the tape, since the **init** option is not specified.

2. For UNIX:

```
dump database pubs2
    to "/dev/rmt4" at REMOTE_BKP_SERVER
    stripe on "/dev/nrmt5" at REMOTE_BKP_SERVER
    stripe on "/dev/nrmt0" at REMOTE_BKP_SERVER
with retaindays = 14
```

   For OpenVMS:

```
dump database pubs2
    to "MTA0:" at REMOTE_BKP_SERVER
    stripe on "MTA1:" at REMOTE_BKP_SERVER
    stripe on "MTA2:" at REMOTE_BKP_SERVER
```

Dumps the pubs2 database, using the REMOTE_BKP_SERVER Backup Server. The command names three dump devices, so the Backup Server dumps approximately one third of the database to each device. This command appends the dump to existing files on the tapes. On UNIX systems, the **retaindays** option specifies that the tapes cannot be overwritten for 14 days. (OpenVMS systems do not use the **retaindays** option; they always create new versions of files.)

3. For UNIX:

```
dump database pubs2
    to "/dev/nrmt0"
    with init
```

For OpenVMS:

```
dump database pubs2
    to "MTA0:"
    with init
```

The **init** option initializes the tape volume, overwriting any existing files.

4. For UNIX:

```
dump database pubs2
    to "/dev/nrmt0"
    with unload
```

For OpenVMS:

```
dump database pubs2
  to "MTA0:"
    with unload
```

Rewinds the dump volumes upon completion of the dump.

5. For UNIX:

```
dump database pubs2
    to "/dev/nrmt0"
    with notify = client
```

For OpenVMS:

```
dump database pubs2
  to "MTA0:"
    with notify = client
```

The **notify** clause sends Backup Server messages requesting volume changes to the client which initiated the dump request, rather than sending them to the default location, the console of the Backup Server machine.

Comments

Commands Used to Back Up Databases

Table 1-12 describes the commands and system procedures used to back up databases:

Use This Command	To
dump database	Make routine dumps of the entire database, including the transaction log
dump transaction	Make routine dumps of the transaction log, then truncate the inactive portion
dump transaction with no_truncate	Dump the transaction log after failure of a database device
dump transaction with truncate_only	Truncate the log without making a backup
dump database	Then copy the entire database
dump transaction with no_log	Truncate the log after your usual method fails due to insufficient log space
dump database	Then copy the entire database
sp_volchanged	Respond to the Backup Server's volume change messages

Table 1-12: Commands used to back up databases and logs

dump database *Restrictions*

- You cannot load a dump that was made on a different platform or with an earlier version of SQL Server.
- If a database has cross-database referential integrity constraints, the *sysreferences* system table stores the **name**—not the ID number—of the external database. SQL Server cannot guarantee referential integrity if you use **load database** to change the database name or to load it onto a different server.

◆ ***WARNING!***

Before dumping a database in order to load it with a different name or move it to another SQL Server, use alter table to drop all external referential integrity constraints.

- You cannot use the **dump database** command in a user-defined transaction.
- If you issue a **dump database** command on a database where a **dump transaction** is already in progress, the **dump database** command sleeps until the transaction dump completes.
- When using quarter-inch cartridge tape, you can dump only one database or transaction log per tape.

Scheduling Dumps

- SQL Server database dumps are **dynamic**—they can take place while the database is active. Because it may slow the system down slightly, so you may want to run **dump database** when the database is not being heavily updated.
- **Back up the *master* database regularly and frequently.** In addition to your regular backups, dump *master* after each **create database**, **alter database**, and **disk init** command is issued.
- Back up the *model* database each time you make a change to the database.
- Use **dump database** immediately after creating a database, to make a copy of the entire database. You cannot run **dump transaction** on a new database until you have run **dump database**.
- Each time you add or remove a cross-database constraint, or drop a table that contains a cross-database constraint, dump **both** of the affected databases.

◆ ***WARNING!***

Loading earlier dumps of these databases could cause database corruption.

- Develop a regular schedule for backing up user databases and their transaction logs.
- Use thresholds to automate backup procedures. To take advantage of SQL Server's last-chance threshold, create user databases with a separate log segment. See the *System Administration Guide* for more information about thresholds.

Dumping the System Databases

- The *master*, *model*, and *sybsystemprocs* databases do not have a separate segment for their transaction logs. Use **dump transaction**

with truncate_only to purge the log, then **dump database** to back up the database.

- The backups of the *master* database are needed for recovery procedures in case of a failure that affects the *master* database. See the *System Administration Guide* for step-by-step instructions for backing up and restoring the *master* database.

Specifying Dump Devices

- You can specify the dump device as a literal, a local variable, or a parameter to a stored procedure.
- You cannot dump to the "null device" (on UNIX, */dev/null*; on OpenVMS, any device name beginning with NL).
- You can specify a local dump device as:
  - A logical device name from the *sysdevices* system table
  - An absolute pathname
  - A relative pathname

  The Backup Server resolves relative pathnames using SQL Server's current working directory.
- When dumping across the network, you must specify the absolute pathname of the dump device. The pathname must be valid on the machine on which the Backup Server is running. If the name includes any characters except letters, numbers or the underscore (_), you must enclose it in quotes.
- Ownership and permissions problems on the dump device may interfere with use of **dump** commands. The **sp_addumpdevice** procedure adds the device to the system tables, but does not guarantee that you can dump to that device or create a file as a dump device.
- You can run more than one dump (or load) at the same time as long as they use different dump devices.

Backup Servers

- You must have a Backup Server running on the same machine as your SQL Server. (On OpenVMS systems, the Backup Server can be running in the same cluster as the SQL Server, as long as all database devices are visible to both.) The Backup Server must be listed in the *master..sysservers* table. This entry is created during installation or upgrade, and should not be deleted.

- If your backup devices are located on another machine, so that you dump across a network, you must also have a Backup Server installed on the remote machine.

Dump Files

- Dumping a database with the **init** option overwrites any existing files on the tape or disk.
- Dump file names identify which database was dumped and when the dump was made. If you do not specify a file name, Backup Server creates a default file name by concatenating the:
  - Last seven characters of the database name
  - Two digit year number
  - Three-digit day of the year (1 through 366)
  - Hexadecimal-encoded time at which the dump file was created

  For example, the file *cations930590E100* contains a copy of the *publications* database made on the fifty-ninth day of 1993:

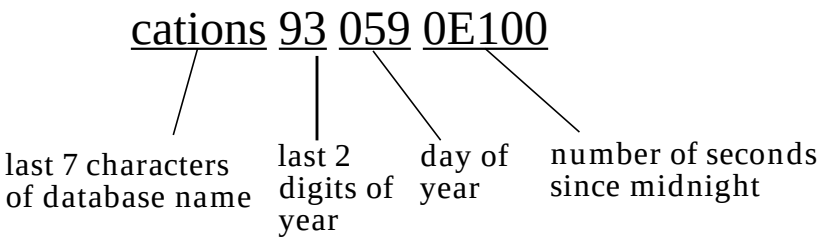


Figure 1-1: File naming convention for database dumps

- The Backup Server sends the dump file name to the location specified by the **with notify** clause. Before storing a backup tape, the operator should label it with the database name, file name, date, and other pertinent information. When loading a tape without an identifying label, use the **with headeronly** and **with listonly** options to determine the contents.

Volume Names

- Dump volumes are labeled according to the ANSI tape-labeling standard. The label includes the logical volume number and the position of the device within the stripe set.
- During loads, Backup Server uses the tape label to verify that volumes are mounted in the correct order. This allows you to load from a smaller number of devices than you used at dump time.

➤ ***Note***

When dumping and loading across the network, you must specify the same number of stripe devices for each operation.

Changing Dump Volumes

- On OpenVMS systems, the operating system requests a volume change when it detects the end of a volume or when the specified drive is off-line. After mounting another volume, the operator uses the REPLY command to reply to these messages.
- On UNIX systems, the Backup Server requests a volume change when the tape capacity has been reached. After mounting another volume, the operator notifies the Backup Server by executing the **sp_volchanged** system procedure on any SQL Server that can communicate with the Backup Server.
- If the Backup Server detects a problem with the currently mounted volume, it requests a volume change by sending messages to either the client or its operator console. The operator responds to these messages with the **sp_volchanged** system procedure.

Appending to/Overwriting a Volume

- By default (**noinit**), Backup Server writes successive dumps to the same tape volume, making efficient use of high-capacity tape media. Data is added following the last end-of-tape mark. New dumps can be appended only to the last volume of a multi-volume dump. Before writing to the tape, Backup Server verifies that the first file has not yet expired. If the tape contains non-Sybase data, Backup Server rejects it to avoid destroying potentially valuable information.
- Use the **init** option to reinitialize a volume. If you specify **init**, Backup Server overwrites any existing contents, even if the tape contains non-Sybase data, the first file has not yet expired, or the tape has ANSI access restrictions.
- Figure 1-2 illustrates how to dump three databases to a single volume using:
  - **init** to initialize the tape for the first dump,
  - **noinit** (the default) to append subsequent dumps

- **unload** to rewind and unload the tape after the last dump

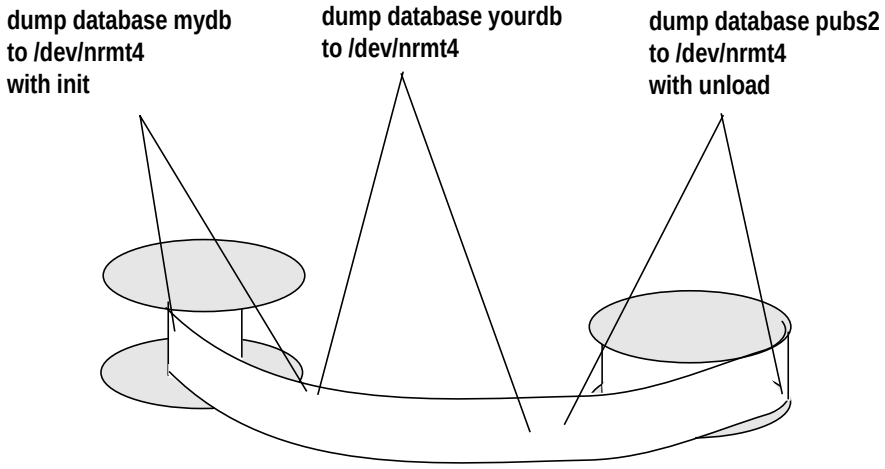


Figure 1-2: Dumping several databases to the same volume

Dumping Databases Whose Devices Are Mirrored

- At the beginning of a **dump database**, SQL Server passes Backup Server the primary device name of all database and log devices. If the primary device has been unmirrored, SQL Server passes the name of the secondary device instead. If any named device fails before the Backup Server completes its data transfer, SQL Server aborts the dump.
- If you attempt to unmirror any of the named database devices while a **dump database** is in progress, SQL Server displays a message. The user executing the **disk unmirror** command can abort the dump or defer the **disk unmirror** until after the dump completes.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

Permissions

Only the System Administrator, the Database Owner, and users with the Operator role can execute **dump database.**

See Also

Commands	**dump transaction, load database, load transaction**
Topics	Roles
System procedures	**sp_addthreshold, sp_addumpdevice, sp_dropdevice, sp_dropthreshold, sp_helpdevice, sp_helpdb, sp_helpthreshold, sp_logdevice, sp_spaceused, sp_volchanged**

dump transaction

Function

Makes a copy of a transaction log and removes the inactive portion.

Syntax

To make a routine log dump:

```
dump tran[saction] database_name
   to stripe_device [ at backup_server_name ]
       [density = density_value,
        blocksize = number_bytes,
        capacity = number_kilobytes,
        dumpvolume = volume_name,
        file = file_name]
   [stripe on stripe_device [ at backup_server_name ]
       [density = density_value,
        blocksize = number_bytes,
        capacity = number_kilobytes,
        dumpvolume = volume_name,
        file = file_name]]
   [[stripe on stripe_device [ at backup_server_name ]
       [density = density_value,
        blocksize = number_bytes,
        capacity = number_kilobytes,
        dumpvolume = volume_name,
        file = file_name] ]...]
   [with {
       density = density_value,
       blocksize = number_bytes,
       capacity = number_kilobytes,
       dumpvolume = volume_name,
       file = file_name,
       [dismount | nodismount],
       [nounload | unload],
       retaindays = number_days,
       [noinit | init],
       notify = {client | operator_console}}]
```

To truncate the log without making a backup copy:

```
dump tran[saction] database_name
   with truncate_only
```

To truncate a log that is filled to capacity. **Use only as a last resort**:

```
dump tran[saction] database_name
   with no_log
```

To back up the log after a database device fails:

```
dump tran[saction] database_name
   to stripe_device [ at backup_server_name ]
       [density = density_value,
        blocksize = number_bytes,
        capacity = number_kilobytes,
        dumpvolume = volume_name,
        file = file_name]
   [stripe on stripe_device [ at backup_server_name ]
       [density = density_value,
        blocksize = number_bytes,
        capacity = number_kilobytes,
        dumpvolume = volume_name,
        file = file_name]]
   [[stripe on stripe_device [ at backup_server_name ]
       [density = density_value,
        blocksize = number_bytes,
        capacity = number_kilobytes,
        dumpvolume = volume_name,
        file = file_name] ]...]
   [with {
       density = density_value,
       blocksize = number_bytes,
       capacity = number_kilobytes,
       dumpvolume = volume_name,
       file = file_name,
       [dismount | nodismount],
       [nounload | unload],
       retaindays = number_days,
       [noinit | init],
       no_truncate,
       notify = {client | operator_console}}]
```

Keywords and Options

database_name – is the name of the database from which you are copying data. The name can be given as a literal, a local variable, or a parameter to a stored procedure.

truncate_only – removes the inactive part of the log **without making a backup copy**. Use on databases without separate log segments, Do not specify a dump device or Backup Server name.

no_log – removes the inactive part of the log **without making a backup copy and without recording the procedure in the transaction log**. Use **no_log** only when you have totally run out of log space and cannot run your usual **dump transaction** command.

to *stripe_device* – is the device to which data is being dumped. See "Specifying Dump Devices" for information about what form to use when specifying a dump device. See the *System Administration Guide Supplement* for your platform for a list of supported dump devices.

at *backup_server_name* – is the name of the Backup Server. This parameter is optional when dumping to the default Backup Server. When dumping across the network, specify the *network name* of a remote Backup Server running on the machine to which the dump device is attached. For platforms that use *interfaces* files, the *backup_server_name* must appear in the *interfaces* file.

density = *density_value* – overrides the default density for a tape device. **Use this option only when reinitializing a volume on OpenVMS systems**. Valid densities are 800, 1600, 6250, 6666, 10000, and 38000. Not all values are valid for every tape drive; use the correct density for your tape drive.

blocksize = *number_bytes* – overrides the default blocksize for a dump device. **(Wherever possible, use the default blocksize**; it is the "best blocksize for your system.) The blocksize must be at least one database page (2048 bytes for most systems, 4096 for Stratus) and must be an exact multiple of the database page size. On OpenVMS systems, blocksize cannot exceed 16,384 bytes.

capacity = *number_kilobytes* – is the maximum amount of data that the device can write to a single tape volume. The capacity must be at least five database pages, and should be slightly less than the recommended capacity for your device.

OpenVMS systems write until they reach the physical end-of-tape marker, then send a volume change request. On UNIX platforms that cannot reliably detect the end-of-tape marker, you must indicate how many kilobytes can be dumped to the tape. You **must** supply a **capacity** for dump devices specified as a physical pathname. If a dump device is specified as a logical device name, the Backup Server uses the *size* parameter stored in the *sysdevices* system table unless you specify a capacity.

dumpvolume = *volume_name* – establishes the name that is assigned to the volume. The Backup Server writes the *volume_name* in the ANSI tape label when overwriting an existing dump, dumping to a brand new tape, or dumping to a tape whose contents are not recognizable. The **load transaction** command checks the label and generates an error message if the wrong volume is loaded.

stripe on *stripe_device* – is an additional dump device. You can use up to 32 devices, including the device named in the **to** *stripe_device* clause. The Backup Server splits the log into approximately equal portions, and sends each portion to a different device. Dumps are made concurrently on all devices, reducing the time and the number of volume changes required. See ''Specifying Dump Devices'' for information about how to specify a dump device.

dismount | **nodismount** – **on platforms that support logical dismount** (such as OpenVMS), determines whether tapes remain mounted. By default, all tapes used for a dump are dismounted when the dump completes. Use **nodismount** to keep tapes available for additional dumps or loads.

nounload | **unload** – determines whether tapes rewind after the dump completes. By default, tapes do not rewind, allowing you to make additional dumps to the same tape volume. Specify **unload** for the last dump file to be added to a multi-dump volume. This rewinds and unloads the tape when the dump completes.

retaindays = *number_days* – **on UNIX systems**, specifies the number of days that Backup Server protects you from overwriting a dump. **This option is meaningful for disk, quarter-inch cartridge, and single-file media. On multi-file media, this option is meaningful for all volumes but the first.** If you try to overwrite a dump before it expires, Backup Server requests confirmation before overwriting the unexpired volume.

The *number_days* must be a positive integer, or 0 for dumps you can overwrite immediately. If you do not specify a **retaindays** value, Backup Server uses the server-wide "tape retention" value, set by **sp_configure.**

noinit | **init** – determines whether the dump is appended to existing dump files or reinitializes (overwrites) the tape volume. By default, dumps are appended following the last end-of-tape mark, allowing you to dump additional databases to the same volume. New dumps can be appended only to the last volume of a multi-volume dump. Use **init** for the first database you dump to a tape, to overwrite its contents.

file = *file_name* – is the name of the dump file. The name cannot exceed 17 characters and must conform to operating system conventions for file names. If you do not specify a file name, Backup Server creates a default file name. For more information, see ''Dump Files''.

no_truncate – dumps a transaction log **even if the disk containing the data segments for a database is inaccessible**, using a pointer to the transaction log in the *master* database. The **with no_truncate** option provides up-to-the-minute log recovery when the transaction log resides on an undamaged device, and the *master* database and user databases reside on different physical devices.

notify = {**client** | **operator_console**} – overrides the default message destination.

- On operating systems (such as OpenVMS) that offer an operator terminal feature, volume change messages are always sent to the operator terminal on the machine on which the Backup Server is running. Use **client** to route other Backup Server messages to the terminal session that initiated the **dump database**.
- On operating systems (such as UNIX) that do not offer an operator terminal feature, messages are sent to the client that initiated the **dump database**. Use **operator_console** to route messages to the terminal on which the Backup Server is running.

Examples

1. For UNIX:

```
dump transaction pubs2
  to "/dev/nrmt0"
```

   For OpenVMS:

```
dump database pubs2
  to "MTA0:"
```

   Dumps the transaction log to a tape, appending it to the files on the tape, since the **init** option is not specified.

2. For UNIX:

```
dump transaction mydb
    to "/dev/nrmt4" at REMOTE_BKP_SERVER
    stripe on "/dev/nrmt5" at REMOTE_BKP_SERVER
with init, retaindays = 14
```

   For OpenVMS:

```
dump transaction mydb
    to "MTA0:" at REMOTE_BKP_SERVER
    stripe on "MTA1:" at REMOTE_BKP_SERVER
with init
```

   Dumps the transaction log for the *mydb* database, using the Backup Server REMOTE_BKP_SERVER. The Backup Server dumps approximately half of the log to each of the two devices. The **init**

option overwrites any existing files on the tape. On UNIX systems, the **retaindays** option specifies that the tapes cannot be overwritten for fourteen days. (OpenVMS systems do not use **retaindays**; they always create new versions of dump files.)

Comments

Commands Used to Back Up Databases

Table 1-12 describes the commands and system procedures used to back up databases:

Use This Command	To
dump database	Make routine dumps of the entire database, including the transaction log
dump transaction	Make routine dumps of the transaction log (the system table, *syslogs*), then truncate the inactive portion
dump transaction with no_truncate	Dump the transaction log after failure of a database device
dump transaction with truncate_only	Truncate the log without making a backup
dump database	Then copy the entire database
dump transaction with no_log	Truncate the log after your usual method fails due to insufficient log space
dump database	Then copy the entire database
sp_volchanged	Respond to the Backup Server's volume change messages

Table 1-13: Commands used to back up logs and databases

dump transaction *Restrictions*

- You cannot load a dump that was made on a different platform or with an earlier version of SQL Server.
- You cannot dump to the "null device" (on UNIX, */dev/null*; on OpenVMS, any device name beginning with NL).
- You cannot use the **dump transaction** command in a transaction.
- When using quarter-inch cartridge tape, you can dump only one database or transaction log per tape.

- You cannot issue a **dump transaction** to a device while the **trunc log on chkpt** database option is enabled, or after enabling **select into/bulk copy** and making unlogged changes to the database with **select into**, "fast" bulk copy operations, or default unlogged **writetext** operations. Use **dump database** instead.

◆ ***WARNING!***

Never modify the log table *syslogs* with a *delete*, *update*, or *insert* command.

- If a database does not have a separate log segment, you cannot use **dump transaction** to copy the log and truncate it.
- If a user or threshold procedure issues a **dump transaction** command on a database where a **dump database** or another **dump transaction** is in progress, the second command sleeps until the first completes.
- To restore a database, use **load database** to load the most recent database dump, then **load transaction** to load each subsequent transaction log dump **in the order in which it was made**.
- Each time you add or remove a cross-database constraint, or drop a table that contains a cross-database constraint, dump **both** of the affected databases.

◆ ***WARNING!***

Loading earlier dumps of these databases could cause database corruption.

Copying the Log After Device Failure: **with no_truncate**

- After device failure, use **dump transaction with no_truncate** to copy the log without truncating it. You can use this option only if your log is on a separate segment and your *master* database is accessible.
- The backup created by **dump transaction with no_truncate** is the most recent dump for your log. When restoring the database, load this dump last.

Databases Without Separate Log Segments: **with truncate_only**

- When a database does not have a separate log segment, use **dump transaction with truncate_only** to remove committed transactions from the log without making a backup copy.

◆ ***WARNING!***

***dump transaction with truncate_only* provides no means to recover your databases. Run *dump database* at the earliest opportunity to ensure recoverability.**

- Use **with truncate_only** on the *master*, *model*, and *sybsystemprocs* databases, which do not have a separate log segment.
- You can also use this option on very small databases, which can store the transaction log and data on the same device.
- Mission-critical user databases should have a separate log segment. Use the **log on** clause of **create database** to create a database with a separate log segment, or **alter database** and **sp_logdevice** to transfer the log to a separate device.

***When All Else Fails:* with no_log**

- Use **dump transaction with no_log** only as a last resort, after your usual method of dumping the transaction log (**dump transaction** or **dump transaction with truncate_only**) fails because of insufficient log space.
- **dump transaction with no_log** truncates the log without logging the dump transaction event. Because it copies no data, it requires only the name of the database.
- Every use of **dump transaction with no_log** is considered an error, and is recorded in the server's error log.

◆ ***WARNING!***

***dump transaction with no_log* provides no means to recover your databases. Run *dump database* at the earliest opportunity to ensure recoverability.**

- If you have created your databases with separate log segments, written a last-chance threshold procedure that dumps your transaction log often enough, and allocated enough space to your log and database, you should not have to use this option. If you must use **with no_log**, increase the frequency of your dumps and the amount of log space.

Scheduling Dumps

- Transaction log dumps are **dynamic**—they can take place while the database is active. Because it may slow the system down slightly, so you may want to run dumps when the database is not being heavily updated.
- Use **dump database** immediately after creating a database, to make a copy of the entire database. You cannot run **dump transaction** on a new database until you have run **dump database**.
- Develop a regular schedule for backing up user databases and their transaction logs.
- **dump transaction** uses less storage space and takes less time than **dump database.** Transaction log dumps are typically made more frequently than database dumps.

Using Thresholds to Automate dump transaction

- Use thresholds to automate backup procedures. To take advantage of SQL Server's last-chance threshold, create user databases with a separate log segment.
- When space on the log segment falls below the last-chance threshold, SQL Server executes the last-chance threshold procedure. Including a **dump transaction** command in your last-chance threshold procedure helps protect you from running out of log space. For more information, see **sp_thresholdaction** in Volume 2.
- You can use **sp_addthreshold** to add a second threshold to monitor log space. For more information about thresholds, see the *System Administration Guide.*

Specifying Dump Devices

- You can specify the dump device as a literal, a local variable, or a parameter to a stored procedure.
- You can specify a local dump device as:
  - A logical device name from the *sysdevices* system table
  - An absolute pathname
  - A relative pathname

  The Backup Server resolves relative pathnames using SQL Server's current working directory.

- When dumping across the network, you must specify the absolute pathname of the dump device. The pathname must be valid on the machine on which the Backup Server is running. If the name includes any characters except letters, numbers or the underscore (_), you must enclose it in quotes.
- Ownership and permissions problems on the dump device may interfere with use of **dump** commands. The **sp_addumpdevice** procedure adds the device to the system tables, but does not guarantee that you can dump to that device or create a file as a dump device.
- You can run more than one dump (or load) at the same time as long as they use different dump devices.

Backup Servers

- You must have a Backup Server running on the same machine as your SQL Server. (On OpenVMS systems, the Backup Server can be running in the same cluster as the SQL Server, as long as all database devices are visible to both.) The Backup Server must be listed in the *master..sysservers* table. This entry is created during installation or upgrade, and should not be deleted.
- If your backup devices are located on another machine, so that you dump across a network, you must also have a Backup Server installed on the remote machine.

Dump Files

- Dumping a log with the **init** option overwrites any existing files on the tape or disk.
- Dump file names identify which database was dumped and when the dump was made. If you do not specify a file name, Backup Server creates a default file name by concatenating the:
  - Last seven characters of the database name
  - Two digit year number
  - Three-digit day of the year (1 through 366)
  - Hexadecimal-encoded time at which the dump file was created

For example, the file *cations930590E100* contains a copy of the *publications* database made on the fifty-ninth day of 1993:

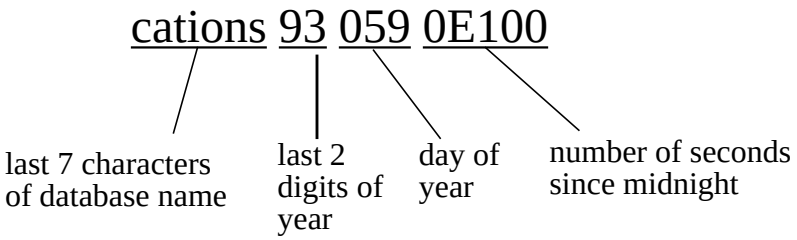


Figure 1-3: File naming convention for transaction log dumps

- The Backup Server sends the dump file name to the location specified by the **with notify** clause. Before storing a backup tape, the operator should label it with the database name, file name, date, and other pertinent information. When loading a tape without an identifying label, use the **with headeronly** and **with listonly** options to determine the contents.

Volume Names

- Dump volumes are labeled according to the ANSI tape-labeling standard. The label includes the logical volume number and the position of the device within the stripe set.
- During loads, Backup Server uses the tape label to verify that volumes are mounted in the correct order. This allows you to load from a smaller number of devices than you used at dump time.

➤ ***Note***

When dumping and loading across the network, you must specify the same number of stripe devices for each operation.

Changing Dump Volumes

- On OpenVMS systems, the operating system requests a volume change when it detects the end of a volume or when the specified drive is off-line. After mounting another volume, the operator uses the REPLY command to reply to these messages.
- On UNIX systems, the Backup Server requests a volume change when the tape capacity has been reached. After mounting another volume, the operator notifies the Backup Server by

executing the **sp_volchanged** system procedure on any SQL Server that can communicate with the Backup Server.

- If the Backup Server detects a problem with the currently mounted volume (for example, the wrong volume is mounted), it requests a volume change by sending messages to either the client or its operator console. The operator responds to these messages with the **sp_volchanged** system procedure.

Appending to/Overwriting a Volume

- By default (**noinit**), Backup Server writes successive dumps to the same tape volume, making efficient use of high-capacity tape media. Data is added following the last end-of-tape mark. New dumps can be appended only to the last volume of a multi-volume dump. Before writing to the tape, Backup Server verifies that the first file has not yet expired. If the tape contains non-Sybase data, Backup Server rejects it to avoid destroying potentially valuable information.
- Use the **init** option to reinitialize a volume. If you specify **init**, Backup Server overwrites any existing contents, even if the tape contains non-Sybase data, the first file has not yet expired, or the tape has ANSI access restrictions.
- Figure 1-4 illustrates how to dump three transaction logs to a single volume. Use:
  - **init** to initialize the tape for the first dump.
  - **noinit** (the default) to append subsequent dumps
  - **unload** to rewind and unload the tape after the last dump

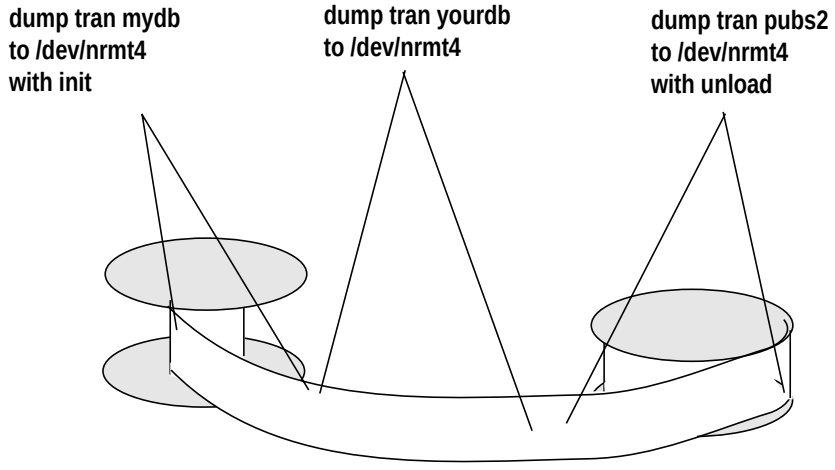


Figure 1-4: Dumping three transaction logs to a single volume

Dumping Logs Stored on Mirrored Devices

- At the beginning of a **dump transaction**, SQL Server passes the primary device name of each logical log device to the Backup Server. If the primary device has been unmirrored, SQL Server passes the name of the secondary device instead. If the named device fails before Backup Server completes its data transfer, SQL Server aborts the dump.
- If you attempt to unmirror a named log device while a **dump transaction** is in progress, SQL Server displays a message. The user executing the **disk unmirror** command can abort the dump or defer the **disk unmirror** until after the dump completes.
- **dump transaction with truncate_only** and **dump transaction with no_log** do not use the Backup Server. These commands are not affected when a log device is unmirrored, either by a device failure or by a **disk unmirror** command.
- **dump transaction** copies only the log segment. It is not affected when a data-only device is unmirrored, either by a device failure or by a **disk unmirror** command.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

Permissions

Only System Administrators, users who have been granted the Operator role, and the Database Owner can execute **dump transaction**.

See Also

Commands	**dump database, load database, load transaction**
Topics	Roles
System procedures	**sp_addumpdevice, sp_dboption, sp_dropdevice, sp_helpdevice, sp_logdevice, sp_volchanged**

execute

Function

Runs a system procedure or a user-defined stored procedure.

Syntax

```
[execute] [@return_status = ]
   [[[server.]database.]owner.]procedure_name[;number]
       [[@parameter_name =] value |
           [@parameter_name =] @variable [output]
       [,[@parameter_name =] value |
           [@parameter_name =] @variable [output]...]]
   [with recompile]
```

Keywords and Options

execute – is used to execute a stored procedure. It is only necessary if the stored procedure call is **not** the first statement in a batch.

@return_status – is an optional integer variable that stores the return status of a stored procedure. It must be declared in the batch or stored procedure before it is used in an execute statement.

procedure_name – is the name of a procedure that has been defined with a **create procedure** statement. You can execute a procedure in another database as long as you are its owner or have permission to execute it in that database. You can execute a procedure on another SQL Server as long as you have permission to use that server and to execute the procedure in that database.

If you specify a server name, but do not specify a database name, SQL Server looks for the procedure in your default database. The owner name is optional only if the Database Owner ("dbo") owns the procedure or if you own it.

;number – is an optional integer used to group procedures of the same name so that they can be dropped together with a single **drop procedure** statement. Procedures used in the same application are often grouped this way. For example, if the procedures used with the application *orders* are named *orderproc;1*, *orderproc;2*, and so on, the statement:

```
drop proc orderproc
```

drops the entire group. Once procedures have been grouped, individual procedures within the group cannot be dropped. For example, you cannot execute the statement:

```
drop procedure orderproc;2
```

parameter_name – is the name of an argument to the procedure, as defined in the **create procedure** statement. Parameter names must be preceded by the "@" symbol.

If the "*@parameter_name = value*" form is used, parameter names and constants need not be supplied in the order defined in the **create procedure** statement. However, if this form is used for any parameter, it must be used for all subsequent parameters.

value – is the value of the parameter or argument to the procedure. If you do not use parameter names, parameter values must be supplied in the order defined in the **create procedure** statement.

If a default is defined in the **create procedure** statement, a user can execute the procedure without giving a parameter. The default must be a constant. It can include the wildcard characters (%, _ , [], and [^]) if the procedure uses the parameter name with the keyword **like**. See Example 2 in the **create procedure** section.

The default can be NULL. Usually, the procedure definition specifies what action should be taken if a parameter value is NULL.

@variable – is the name of a variable used to store a return parameter.

output – indicates that the stored procedure is to return a return parameter. The matching parameter in the stored procedure must also have been created with the keyword **output**.

with recompile – forces compilation of a new plan. Use this option if the parameter you're supplying is atypical or if the data has significantly changed. The changed plan is used on subsequent executions.

Examples

1.
```
execute showind titles
```

or:

```
exec showind @tabname = titles
```

or, if this is the only statement in a batch or file:

```
showind titles
```

All three examples above execute the stored procedure *showind* with a parameter value *titles*.

2.
```
declare @retstat int
execute @retstat = GATEWAY.pubs.dbo.checkcontract
"409-56-4008"
```

   Executes the stored procedure *checkcontract* on the remote server GATEWAY. Stores the return status indicating success or failure in *@retstat*.

3.
```
declare @percent int
select @percent = 10
execute roy_check "BU1032", 1050, @pc = @percent
output
select Percent = @percent
```

   Executes the stored procedure *roy_check*, passing three parameters. The third parameter, *@pc*, is an **output** parameter. After execution of the procedure, the return value is available in the variable *@percent*.

Comments

- There are two ways to supply parameters—by position, or by using:

  parameter_name = value

  If you use the second form, you do not have to supply the parameters in the order defined in the **create procedure** statement.

  If you are using the **output** keyword and intend to use the return parameters in additional statements in your batch or procedure, the value of the parameter must be passed as a variable, for example:

  parameter_name = @variable_name

- You cannot use *text* and *image* columns as parameters to stored procedures or as values passed to parameters.
- It is an error to execute a procedure specifying **output** for a parameter that is not defined as a return parameter in the **create procedure** statement.
- You cannot pass constants to stored procedures using **output**; the return parameter requires a variable name. You must declare the variable's datatype and assign it a value before executing the procedure. Return parameters cannot have a datatype of *text* or *image*.

- It is not necessary to use the keyword **execute** if the statement is the first one in a batch. A batch is a segment of an input file terminated by the word "go" on a line by itself.
- Since the execution plan for a procedure is stored the first time it is run, subsequent run time is much shorter than for the equivalent set of stand-alone statements.
- Nesting occurs when one stored procedure calls another. The nesting level is incremented when the called procedure begins execution and it is decremented when the called procedure completes execution. Exceeding the maximum of 16 levels of nesting causes the transaction to fail. The current nesting level is stored in the *@@nestlevel* global variable.
- Return values 0 and -1 through -14 are currently used by SQL Server to indicate the execution status of stored procedures. Values from -15 to -99 are reserved for future use. See **return** for a list of values.
- Parameters are not part of transactions, so if a parameter is changed in a transaction which is later rolled back, its value does not revert to its previous value. The value that is returned to the caller is always the value at the time the procedure returns.
- If you use **select** * in your **create procedure** statement, the procedure does not pick up any new columns you may have added to the table (even if you use the with recompile option to execute). You must **drop** the procedure and recreate it.
- Commands executed via remote procedure calls cannot be rolled back.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

Permissions

execute permission defaults to the owner of the stored procedure, who can transfer it to other users.

See Also

Commands	**create procedure**, **drop procedure**, **return**
Topics	Parameters, Variables (Local and Global), Wildcard Character
System procedures	**sp_depends**, **sp_helptext**

fetch

Function

Returns a row or a set of rows from a cursor result set.

Syntax

```
fetch cursor_name [ into fetch_target_list ]
```

Parameters

cursor_name – the name of the cursor

into *fetch_target_list* – is a comma-separated list of parameters or local variables into which cursor results are placed. The parameters and variables must be declared prior to the **fetch**.

Examples

1. ```
   fetch authors_crsr
   ```

   Returns a row of information from the cursor result set defined by the *authors_crsr* cursor.

2. ```
   fetch pubs_crsr into @name, @city, @state
   ```

   Returns a row of information from the cursor result set defined by the *pubs_crsr* cursor into the variables *@name*, *@city*, and *@state*.

Comments

Restrictions

- Before you can use **fetch**, you must declare the cursor and **open** it.
- The *cursor_name* cannot be a Transact-SQL parameter or local variable.
- You cannot **fetch** a row that has already been fetched. There is no way to backtrack through the result set. But you can close and re-open the cursor to create the cursor result set again and start from the beginning.
- SQL Server expects a one-to-one correspondence between the variables in the *fetch_target_list* and the target list expressions specified by the *select_statement* that defines the cursor. The datatypes of the variables or parameters must be compatible with the datatypes of the columns in the cursor result set.

- When you set chained transaction mode, SQL Server implicitly begins a transaction with the **fetch** statement if no transaction is currently active. However, this situation occurs only when you set the **close on endtran** option and the cursor remains open after the end of the transaction that initially opened it, since the **open** statement also automatically begins a transaction.

Cursor Position

- After you **fetch** all the rows, the cursor points to the last row of the result set. If you **fetch** again, SQL Server returns a warning through the *@@sqlstatus* variable (described below) indicating there is no more data, and the cursor position moves beyond the end of the result set. You can no longer **update** or **delete** from that current cursor position.
- With **fetch into**, SQL Server does not advance the cursor position if an error occurs because the number of variables in the *fetch_target_list* does not equal the number of target list expressions specified by the query that defines the cursor. However, it does advance the cursor position even if a compatibility error occurs between the datatypes of the variables and the datatypes of the columns in the cursor result set.

Determining How Many Rows Are Fetched

- You can **fetch** one or more rows at a time. Use the **cursor rows** option of the **set** command to specify the number of rows to **fetch**.

Getting Information About Fetches

- The *@@sqlstatus* global variable holds status information (warning exceptions) resulting from the execution of a **fetch** statement. The value of *@@sqlstatus* is 0, 1, or 2.

0	Indicates successful completion of the **fetch** statement.
1	Indicates that the **fetch** statement resulted in an error.
2	Indicates that there is no more data in the result set. This warning can occur if the current cursor position is on the last row in the result set and the client submits a **fetch** statement for that cursor.

Table 1-14: @@sqlstatus values

Only a **fetch** statement can set *@@sqlstatus*. All other statements have no effect on *@@sqlstatus*.

- The *@@rowcount* global variable holds the number of rows returned from the cursor result set to the client up to the last **fetch**. In other words, it represents the total number of rows seen by the client at any one point in time.

  Once all the rows have been read from the cursor result set, *@@rowcount* represents the total number of rows in the cursor results set. Each open cursor is associated with a specific *@@rowcount* variable, which is dropped when you close the cursor. Check *@@rowcount* after a **fetch** to get the number of rows read for the cursor specified in that **fetch**.

Standards and Compliance

Standard	Compliance Level	Comments
SQL89	Compliant	The use of variables in a target list and fetch of multiple rows are Transact-SQL extensions.
SQL92	Entry level compliant	

Permissions

fetch permission defaults to all users.

See Also

Commands	**declare cursor**, **open**, **set**
Topics	Cursors, Transactions

goto Label

Function

Branches to a user-defined label.

Syntax

```
label:
  goto label
```

Examples

```
declare @count smallint
select @count = 1
restart:
  print "yes"
select @count = @count + 1
while @count <=4
  goto restart
```

Comments

- The label name must conform to the rules for identifiers and must be followed by a colon (:) when it is declared. It is not followed by a colon when it is used with **goto**.
- The **goto** is usually made dependent on an **if** or **while** test, or some other condition, in order to avoid an endless loop between **goto** and the label.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

See Also

Commands	if...else, while
Topics	Control-of-Flow Language

grant

Function

Assigns permissions to users.

Syntax

To grant permission to access database objects:

```
grant {all [privileges]| permission_list}
   on { table_name [(column_list)]
        | view_name[(column_list)]
        | stored_procedure_name}
   to {public | name_list | role_name}
   [with grant option]
```

To grant permission to create database objects:

```
grant {all [privileges] | command_list}
   to {public | name_list | role_name}
```

Keywords and Options

all – when used to assign permission to access database objects (the first syntax format), **all** specifies that all permissions applicable to the specified object are granted. All object owners can use **grant all** with an object name to grant permissions on their own objects.

Only a System Administrator or the Database Owner can assign permission to create database objects (the second syntax format). When used by a System Administrator, **grant all** assigns all **create** permissions (**create database**, **create default**, **create procedure**, **create rule**, **create table**, and **create view**). When the Database Owner uses **grant all**, SQL Server grants all **create** permissions except **create database**, and prints an informational message.

permission_list – is a list of object access permissions granted. If more than one permission is listed, separate them with commas. The

following table illustrates the access permissions that can be granted on each type of object

Object	*permission_list* Can Include:
table or view	**select, insert, delete, update, references**
column	**select, update, references** Column names can be specified in either *permission_list* or *column_list* (see example 2).
stored procedure	**execute**

Table 1-15: Object access permissions

command_list – is a list of object creation permissions to be granted. If more than one command is listed, separate them with commas. The command list can include **create database**, **create default**, **create procedure**, **create rule**, **create table**, and **create view**. **create database** permission can only be granted by a System Administrator, and only from within the *master* database.

table_name – is the name of the table on which you are granting permissions. The table must be in your current database. Only one object can be listed for each **grant** statement.

column_list – is a list of columns, separated by commas, to which the permissions apply. If columns are specified, only **select**, **references**, and **update** permissions can be granted.

view_name – is the name of the view on which you are granting permissions. The view must be in your current database. Only one object can be listed for each **grant** statement.

stored _procedure_name – is the name of the stored procedure on which you are granting permissions. The stored procedure must be in your current database. Only one object can be listed for each **grant** statement.

public – is all users. For object access permissions, **public** excludes the object owner. For object creation permissions, **public** excludes the Database Owner (who "owns" object creation permissions within the database). You cannot **grant** permissions **with grant option** to "public" or to other groups or roles.

name_list – is a list of users' database names and / or group names, separated by commas.

role_name – is the name of a SQL Server role. This allows you to grant specific permissions to all users who have been granted a specific role. The roles are **sa_role** (System Administrator), **sso_role** (System Security Officer), and **oper_role** (Operator). Roles are granted to users with **sp_role**.

with grant option - allows the users specified in *name_list* to grant object access permissions to other users. You can only grant permissions **with grant option** to individual users, not to "public" or to a group or role.

Examples

1.
```
grant insert, delete
on titles
to mary, sales
```

   Grants Mary and the "sales" group permission to use the **insert** and **delete** commands on the *titles* table.

2.
```
grant update
on titles (price, advance)
to public
```

   or:

```
grant update (price, advance)
on titles
to public
```

   Two ways to grant **update** permission on the *price* and *advance* columns of the *titles* table to "public" (which includes all users).

3.
```
grant create database, create table
to mary, john
```

   Grants Mary and John permission to use the **create database** and **create table** commands. Because **create database** permission is being granted, this command can only be executed by a System Administrator within the *master* database. Mary and John's **create table** permission applies only to the *master* database.

4.
```
grant all on titles
to public
```

   Grants complete access permissions on the *titles* table to all users.

5. **grant all**
   to public

   Grants all object creation permissions in the current database to all users. If this command is executed by a System Administrator from the *master* database, it includes **create database** permission.

6. **grant update on authors**
   to mary
   with grant option

   Gives Mary permission to use the **update** command on the *authors* table, and to grant that permission to others.

7. **grant select, update on titles(price)**
   to bob
   with grant option

   Gives Bob permission to use the **select** and **update** commands on the *price* column of the *titles* table, and to grant that permission to others.

8. **grant execute on new_sproc**
   to sso_role

   Grants permission to execute the *new_sproc* stored procedure to all System Security Officers.

9. **grant references on titles(price)**
   to james

   Grants James permission to create a referential integrity constraint on another table that refers to the *price* column of the *titles* table.

Comments

- You can substitute the word **from** for **to** in the **grant** syntax.
- *Table 1-16: Command and object permissions* summarizes default permissions on Transact-SQL commands in SQL Server. The user listed under the "Defaults To" heading is the lowest level of user automatically granted permission to execute a command. This user can **grant** or **revoke** the permission if it is transferable. Users at higher levels than the default are either automatically assigned the permission, or (in the case of Database Owners) can get it via the **setuser** command.

  For example, the owner of a database does not automatically receive permissions on objects owned by other users. A Database Owner can always gain any such permission by assuming the identity of the object owner with the **setuser**

command and then issuing the appropriate **grant** or **revoke** statements. System Administrators have permission to access all commands and objects at any time.

The SQL Server installation script assigns a set of permissions to the default group "public". **grant** or **revoke** statements need not be written for these permissions.

Table 1-16 does not include the System Security Officer, who does not have any special permissions on commands and objects, but only on certain system procedures.

Command and Object Permissions

Statement	Defaults To					Can Be Granted/Revoked		
	System Admin.	Operator	Database Owner	Object Owner	Public	Yes	No	N/A
alter database	•					(1)		
alter table				•			•	
begin transaction					•			•
checkpoint			•				•	
commit					•			•
create database	•					•		
create default			•			•		
create index				•			•	
create procedure			•			•		
create rule			•			•		
create table			•		(2)	• (2)		
create trigger				•			•	
create view			•			•		
dbcc			•				•	
delete				• (3)		•		
disk init	•						•	

(1) Transferred with database ownership
(2) Public can create temporary tables, no permission required
(3) If a view, permission defaults to view owner
(4) Defaults to stored procedure owner

(5) Transferred with **select** permission
(6) Transferred with **update** permission
No means use of the command is never restricted
N/A means use of the command is always restricted

Table 1-16: Command and object permissions

Command and Object Permissions								
Statement	Defaults To					Can Be Granted/Revoked		
	System Admin.	Operator	Database Owner	Object Owner	Public	Yes	No	N/A
disk mirror	•							
disk refit	•							
disk reinit	•							
disk remirror	•							
disk unmirror	•						•	
drop (any object)				•			•	
dump database		•	•				•	
dump transaction		•	•				•	
execute				•(4)		•		
grant on object				•		•		
grant command			•			•		
insert				• (3)		•		
kill	•						•	
load database		•	•				•	
load transaction		•	•				•	
print					•			•
raiserror					•			•
readtext				•		(5)		
reconfigure	•						•	
references				•		•		
revoke on object				•			•	
revoke command			•				•	
rollback					•			•
save transaction					•			•
select				• (3)		•		

(1) Transferred with database ownership
(2) Public can create temporary tables, no permission required
(3) If a view, permission defaults to view owner
(4) Defaults to stored procedure owner

(5) Transferred with **select** permission
(6) Transferred with **update** permission
No means use of the command is never restricted
N/A means use of the command is always restricted

Table 1-16: Command and object permissions (continued)

Command and Object Permissions								
Statement	Defaults To					Can Be Granted/Revoked		
	System Admin.	Operator	Database Owner	Object Owner	Public	Yes	No	N/A
set					•			•
setuser			•				•	
shutdown	•						•	
truncate table				•			•	
update				• (3)		•		
update statistics				•			•	
writetext				•		(6)		

(1) Transferred with database ownership
(2) Public can create temporary tables, no permission required
(3) If a view, permission defaults to view owner
(4) Defaults to stored procedure owner

(5) Transferred with **select** permission
(6) Transferred with **update** permission
No means use of the command is never restricted
N/A means use of the command is always restricted

Table 1-16: Command and object permissions (continued)

- You can only **grant** or **revoke** permissions on objects in the current database.
- Before you create a table that includes a referential integrity constraint to reference another user's table, you must be granted **references** permission on that referenced table (see example 10). The table must also include a **unique** constraint or unique index on the referenced columns. See **create table** for more information about referential integrity constraints.
- **grant** and **revoke** commands are order-sensitive. The command that takes effect when there is a conflict is the most recently issued one.
- A user can be granted permission on a view or stored procedure even if he or she has no permissions on objects the procedure or view references. See the *System Administration Guide* for more information on using views and stored procedures as security mechanisms.
- SQL Server grants all users permission to declare cursors, regardless of the permissions defined for the base tables or views referenced in the **declare cursor** statement. Cursors are not defined

as SQL Server objects (such as tables), so no permissions can be applied against a cursor. When a user opens a cursor, SQL Server does determine if that user has **select** permissions on the objects which define that cursor's result set. It checks permissions on each **open** of a cursor.

If the user has permission to access the objects defined by the cursor, SQL Server opens the cursor and allows the user to **fetch** row data through the cursor. SQL Server does not apply permission checking for each **fetch**. However, if the user performs a **delete** or an **update** through that cursor, the regular permission checking applies for deleting and updating the data of objects referenced in the cursor result set.

- Each **grant** or **revoke** statement adds a row to the *sysprotects* system table. However, if a user inherits a particular permission by virtue of being a member of a group, and then the same permission is explicitly granted to the user, no row is added to *sysprotects*. For example, say that "public" has been granted **select** permission on the *phone* column of the *authors* table. John, a member of "public," is then granted **select** permission on all columns of *authors*. The row added to *sysprotects* as a result of the **grant** to John will contain references to all columns of *authors* except for the *phone* column, on which he already had permission.
- You can get information about permission with these system procedures:
  - **sp_helprotect** reports permissions on a database object or on a user.
  - **sp_column_privileges** reports permissions information for one or more columns in a table or view.
  - **sp_table_privileges** reports permissions information for all columns in a table or view.

grant all (*Object Creation Permissions*)

- When used with only user and/or group names (no object names), **grant all** assigns these permissions: **create database**, **create default**, **create procedure**, **create rule**, **create table**, and **create view. create database** permission can be granted only by a System Administrator and only from within the *master* database.
- Only the Database Owner and System Administrator can use the **grant all** syntax without an object name to grant **create** command permissions to users or groups. When the **grant all** command is used by the Database Owner, an informational message is

printed, stating that only a System Administrator can grant **create database** permission. All other permissions in the list above are granted.

- All object owners can use **grant all** with an object name to grant permissions on their own objects. When used with a table or view name plus user and / or group names, **grant all** enables **delete**, **insert**, **select** and **update** permission on the table.

grant with grant option *Rules*

- You cannot grant permissions **with grant option** to "public" or to a group or role.
- In granting permissions, a System Administrator is treated as the object owner. If a System Administrator grants permission on another user's object, the owner's name appears as the grantor in *sysprotects* and in **sp_helprotect** output.
- Information for each **grant** is kept in the system table *sysprotects* with the following exceptions:
  - SQL Server displays an informational message if a specific permission is granted to a user more than once by the same grantor. Only the first **grant** is kept.
  - If two **grant**s are exactly same except that one of them is granted **with grant option**, the **grant with grant option** is kept.
  - If two **grant** statements grant the same permissions on a particular table to a specific user, but the columns specified in the grants are different, SQL Server treats the grants as if they are one statement. For example, the following **grant** statements are equivalent:

```
grant select on titles(price, contract) to keiko
grant select on titles(advance) to keiko

grant select on titles(price, contract, advance)
to keiko
```

grant *and Roles*

- You can use the **grant** command to grant object creation or object access permissions to all users who have been granted a specified role. (See example 8.) This allows you to restrict use of a stored procedure or an object to users who are System Administrators, System Security Officers, or Operators. To grant or revoke roles, use **sp_role**.

However, **grant execute** permission does not prevent users who do not have the specified role from being individually granted permission to execute a stored procedure. If you want to ensure, for example, that only System Security Officers can ever be granted permission to execute a stored procedure, you can use the **proc_role** system function within the stored procedure itself. It checks to see whether the invoking user has the correct role to execute the procedure. See "System Functions" for more information.

- Permissions granted to roles override permissions granted to users or groups. For example, say John has been granted the System Security Officer role, and **sso_role** has been granted permission on the *sales* table. If John's individual permission on *sales* is revoked, he is still able to access *sales* because his role permissions override his individual permissions.

Users and User Groups

- User groups allow you to **grant** or **revoke** permissions to more than one user with a single statement. Each user can only be a member of one group, and is always a member of "public".
- The Database Owner or System Administrator can add new users with **sp_adduser** and create groups with **sp_addgroup**. To allow users with logins on SQL Server to use the database with limited privileges, you can add a "guest" user with **sp_adduser**, and assign limited permissions to "guest". All users with logins can access the database as "guest".
- To remove a user, use **sp_dropuser.** To remove a group, use **sp_dropgroup**.

  To add a new user to a group other than "public", use **sp_adduser.** To change an established user's group, use **sp_changegroup**.

  To display the members of a group, use **sp_helpgroup**.
- When **sp_changegroup** is executed to change group membership, it clears the in-memory protection cache by executing:

  ```
  grant all to null
  ```

  so that the cache can be refreshed with updated information from the *sysprotects* table. If you need to modify *sysprotects* directly, contact Sybase Technical Support.

Standards and Compliance

Standard	Compliance Level	Comments
SQL89	Compliant	Granting permissions to groups is a Transact-SQL extension.
SQL92	Entry level compliant	

Permissions

grant permission defaults to object owners. Those users can grant permission to other users on their own database objects.

See Also

Commands	**revoke, setuser**
System procedures	**sp_addgroup, sp_adduser, sp_changedbowner, sp_changegroup, sp_dropgroup, sp_dropuser, sp_helpgroup, sp_helprotect, sp_helpuser, sp_role**
Catalog stored procedures	**sp_column_privileges, sp_table_privileges**

group by and having Clauses

Function

Used in **select** statements to divide a table into groups.

Syntax

Start of **select** *statement*

```
group by [all] aggregate_free_expression
   [, aggregate_free_expression]...
[having search_conditions]
```

End of **select** *statement*

Keywords and Options

group by – specifies the groups into which the table will be divided, and if aggregate functions are included in the select list, finds a summary value for each group. These summary values appear as new columns in the results, rather than as new rows. You can refer to these new summary columns in the **having** clause.

You can use the following aggregate functions in the select list before **group by** (the expression is usually a column name):

A table can be grouped by any combination of columns—that is, groups can be nested within each other, as in Example 2.

all – a Transact-SQL extension that includes all groups in the results, even those excluded by a **where** clause. For example:

```
select type, avg(price)
from titles
where advance > 7000
group by all type

type
-----------------  ----------
UNDECIDED                NULL
business                 2.99
mod_cook                 2.99
popular_comp            20.00
psychology               NULL
trad_cook               14.99
(6 rows affected)
```

"NULL" in the aggregate column indicates groups that would be excluded by the **where** clause. **having** clauses negate the meaning of **all**.

aggregate_free_expression – is an expression that includes no aggregates. A Transact-SQL extension allows grouping by an aggregate-free expression as well as by a column name.

Note that you cannot group by a column heading or alias. This example is correct:

```
select Price=avg(price), Pay=avg(advance),
Total=price * $1.15
from titles
group by price * $1.15
```

having – sets conditions for the **group by** clause similar to the way that **where** sets conditions for the **select** clause.

having search conditions can include aggregate expressions, otherwise its search conditions are identical to **where** search conditions. Here's an example of a **having** clause with aggregates:

```
select pub_id, total = sum(total_sales)
from titles
where total_sales is not null
group by pub_id
having count(*)>5
```

A **having** clause can include a maximum of 128 search arguments per result table, where the search arguments are in the form:

```
column_name comparison-operator
    constant-expression and ...
```

There is no limit to the number of expressions.

Examples

1.
```
select type, avg(advance), sum(total_sales)
from titles
group by type
```

   Calculates the average advance and the sum of the sales for each type of book.

2.
```
select type, pub_id, avg(advance), sum(total_sales)
from titles
group by type, pub_id
```

   Groups the results by type and then by *pub_id* within each type.

3.
```
select type, avg(price)
from titles
group by type
having type like 'p%'
```

   Calculates results for all groups but displays only groups whose type begins with "p".

4.
```
select pub_id, sum(advance), avg(price)
from titles
group by pub_id
having sum(advance) > $15000
and avg(price) < $10
and pub_id > "0700"
```

   Displays results for groups matching the conditions in the **having** clause.

Comments

- You can use a column name or any expression (except a column heading or alias) after **group by.** You can **group by** a column or expression that does not appear in the select list (a Transact-SQL extension).
- The maximum number of columns or expressions allowed in a **group by** clause is 16.
- Null values in the **group by** column are put into a single group.
- You cannot name *text* or *image* columns in **group by** and **having** clauses.
- You cannot use a **group by** clause in the **select** statement of an updatable cursor.
- Aggregate functions can be used only in the select list or in a **having** clause. They cannot be used in a **where** or **group by** clause.

  Aggregate functions are of two types. Aggregates applied to **all the rows in a table** (producing a single value for the whole table per function) are called **scalar aggregates**. An aggregate function in the select list with no **group by** clause applies to the whole table; it is one example of a scalar aggregate. Aggregates applied to **a group of rows in a specified column or expression** (producing a value for each group per function) are called **vector aggregates**. In either case, the results of the aggregate operations are shown as new columns that the **having** clause can refer to.

  You can nest a vector aggregate inside a scalar aggregate. (See "Aggregates" for more information and an example.)

How group by *and* having *Queries Work*

- The **where** clause excludes rows that do not meet its search conditions.
- The **group by** clause collects the surviving rows into one group for each unique value in the **group by** expression. Omitting **group by** creates a single group for the whole table.
- The **having** clause excludes rows that do not meet its search conditions. **having** only tests rows, but the presence or absence of a **group by** clause may make its behavior appear contradictory:
  - When the query includes **group by**, **having** excludes result rows from groups. This is why **having** seems to operate on groups.
  - When the query has no **group by**, **having** excludes result rows from the (single-group) table. This is why **having** seems to operate on rows. (See Example 6, following.)
- Aggregate functions specified in the select list calculate summary values for each surviving group.

Standard group by *and* having

- For **group by**, **having**, and aggregate functions to produce one row and one summary value per group:
  - Columns in a select list must also be in the **group by** expression or they must be arguments of aggregate functions.
  - A **group by** expression can only contain column names that are in the select list.
  - Columns in a **having** expression must be single-valued—arguments of aggregates, for instance—and they must be in the select list or **group by** clause. A query with a **having** clause **should** have a **group by** clause, but if omitted, all the rows not excluded by the **where** clause are considered to be a single group.

  In non-grouped queries, the principle that "**where** excludes rows" always seems straightforward. In grouped queries, the principle expands to "**where** excludes rows before **group by**, then **having** excludes rows from the display of results."

Transact-SQL Extensions to group by *and* having

- Transact-SQL extensions to standard SQL make displaying data more flexible, by allowing references to columns and expressions that are not used for creating groups or summary calculations:
  - A select list that includes aggregates can include columns that are not arguments of aggregate functions and are not included in the **group by** clause. Example 2, which follows, illustrates how an "extended column" like this affects the display of results.
  - The **group by** clause can include columns or expressions that are not in the select list.
  - The **group by all** clause displays all groups, even those excluded from calculations by a **where** clause. See the earlier example for the keyword **all**.
  - The **having** clause can include columns or expressions not in the select list and not in the **group by** clause. When the Transact-SQL extensions add rows and columns to the display, or **group by** is omitted, query results can be hard to interpret. Use the chart and the examples that follow to help you keep track of how Transact-SQL extensions can affect query results.

  The following examples illustrate the differences between standard **group by** and **having** clauses and Transact-SQL extensions:

1.
```
select type, avg(price)
from titles
group by type
```

```
type
----------------------  -----------
UNDECIDED                      NULL
business                      13.73
mod_cook                      11.49
popular_comp                  21.48
psychology                    13.50
trad_cook                     15.96
(6 rows affected)
```

A standard grouping query.

```
2. select type, title_id, avg(price)
   from titles
   group by type
```

```
 type        title_id
------------ -------- ----------------
business     BU1032              13.73
business     BU1111              13.73
business     BU2075              13.73
business     BU7832              13.73
mod_cook     MC2222              11.49
mod_cook     MC3021              11.49
UNDECIDED    MC3026               NULL
popular_comp PC1035              21.48
popular_comp PC8888              21.48
popular_comp PC9999              21.48
psychology   PS1372              13.50
psychology   PS2091              13.50
psychology   PS2106              13.50
psychology   PS3333              13.50
psychology   PS7777              13.50
trad_cook    TC3218              15.96
trad_cook    TC4203              15.96
trad_cook    TC7777              15.96

(18 rows affected)
```

The Transact-SQL extended column, *title_id* (in the select list, not an aggregate and not in the **group by** clause) causes all qualified rows to display in each qualified group, even though **group by** should produce a single row per group. The average price per group, displayed on each row of each group, is identical to Example 1.

```
3. select pub_id, count(pub_id)
   from publishers
```

```
pub_id
---------- ---------
0736               3
0877               3
1389               3

(3 rows affected)
```

The "standard extension" of omitting the **group by** clause makes *publishers* a single group. The aggregate applies to the (single-group) table.

```
4. select pub_id, count(pub_id)
   from publishers
   where pub_id < "1000"
   group by pub_id

   pub_id
   -------------- ---------
   0736                   1
   0877                   1

   (2 rows affected)
```

In this standard grouping query, two of the three single-row groups contain *pub_id* values that satisfy the **where** clause. The aggregate function correctly counts one qualifying *pub_id* per group.

```
5. select pub_id, count(pub_id)
   from publishers
   where pub_id < "1000"

   pub_id
   -------------- -----------
   0736                    2
   0877                    2
   1389                    2

   (3 rows affected)
```

This example combines two extension effects. By omitting the **group by** clause:

- The table becomes a single group. The aggregate counts two qualified rows.
- *pub_id* becomes a Transact-SQL extended column because it does not appear in a **group by** clause. No **having** clause is present, so all rows in the group are qualified to display.

```
6. select pub_id, count(pub_id)
   from publishers
   having pub_id < "1000"

   pub_id
   -------------- ---------
   0736                   3
   0877                   3
   (2 rows affected)
```

This example illustrates the effects of a **having** clause used without a **group by** clause.

- The table is considered a single group. No **where** clause excludes rows, so all the rows in the group (table) are qualified to be counted.
- The rows in this single-group table are then tested by the **having** clause.
- These combined effects display the two qualified rows.

```
7. select type, price, avg(price)
   from titles
   where price > 10.00
   group by type
```

```
type        price
------------ ---------------- ----------------
business                19.99            17.31
business                11.95            17.31
business                 2.99            17.31
business                19.99            17.31
mod_cook                19.99            19.99
mod_cook                 2.99            19.99
popular_comp            22.95            21.48
popular_comp            20.00            21.48
popular_comp             NULL            21.48
psychology              21.59            17.51
psychology              10.95            17.51
psychology               7.00            17.51
psychology              19.99            17.51
psychology               7.99            17.51
trad_cook               20.95            15.96
trad_cook               11.95            15.96
trad_cook               14.99            15.96

(17 rows affected)
```

The way that Transact-SQL extended columns are handled can make it look as if a query is ignoring a **where** clause. This query computes the average prices using only those rows that satisfy the **where** clause, but also displays rows that don't match the **where** clause.

SQL Server first builds a work table containing just the type and the aggregate values using the **where** clause. This work table is joined back to the *titles* table on the *type* to include the *price* column in the results, but the **where** clause is **not** used in the join. The only row in *titles* which is not in the results is the lone row with *type* = "undecided" and a NULL price, that is, a row for which there were no results in the work table. If you want to

eliminate the rows that have prices less than $10.00 from the results, you must add a **having** clause that repeats the **where** clause:

```
8. select type, price, avg(price)
   from titles
   where price > 10.00
   group by type
   having price > 10.00

   type        price
   ----------- ---------------- ----------------
   business                19.99            17.31
   business                11.95            17.31
   business                19.99            17.31
   mod_cook                19.99            19.99
   popular_comp            22.95            21.48
   popular_comp            20.00            21.48
   psychology              21.59            17.51
   psychology              10.95            17.51
   psychology              19.99            17.51
   trad_cook               20.95            15.96
   trad_cook               11.95            15.96
   trad_cook               14.99            15.96
   (12 rows affected)
```

If you are specifying additional conditions, such as aggregates, in the **having** clause, be sure to also include all conditions specified in the **where** clause. SQL Server will appear to ignore any **where** clause conditions that are missing from the **having** clause.

group by *and* having *and sort orders*

If your server has a case-insensitive sort order installed, **group by** ignores the case of the grouping columns. For example, given this data on a case-insensitive server:

```
   select lname, amount
   from groupdemo

   lname      amount
   ---------- ------------------
   Smith                  10.00
   smith                   5.00
   SMITH                   7.00
   Levi                    9.00
   Lévi                   20.00
```

Grouping by *lname* produces these results:

```
select lname, sum(amount)
from groupdemo
group by lname

lname
---------- ------------------
Levi                     9.00
Lévi                    20.00
Smith                   22.00
```

The same query on a case- and accent- insensitive server produces these results:

```
lname
---------- ------------------
Levi                    29.00
Smith                   22.00
```

Standards and Compliance

Standard	Compliance Level	Comments
SQL89	Compliant	The use within the select list of columns that are not in the **group by** list and have no aggregate functions is a Transact-SQL extension. The use of the **all** keyword is a Transact-SQL extension.
SQL92	Entry level compliant	

See Also

Commands	**compute** Clause, **declare**, **select**, **where** Clause
Functions	Aggregate Functions, Row Aggregate Functions
Topics	Cursors

if...else

Function

Imposes conditions on the execution of a SQL statement. The statement following an **if** keyword and its condition is executed if the condition is satisfied (when the logical expression returns "true"). The optional **else** keyword introduces an alternate SQL statement that executes when the **if** condition is not satisfied (when the logical expression returns "false").

Syntax

```
if logical_expression
   statements
[else
   [if logical_expression]
   statement]
```

Keywords and Options

logical_expression– is an expression (a column name, a constant, any combination of column names and constants connected by arithmetic or bitwise operators, or a subquery) that returns TRUE, FALSE or NULL. If the expression contains a **select** statement, the **select** statement must be enclosed in parentheses.

statements – is either a single SQL statement or a block of statements delimited by **begin** and **end.**

Examples

```
1. if 3 > 2
    print "yes"

2. if exists (select postalcode from authors
    where postalcode = "94705")
    print "Berkeley author"

3. if (select max(id) from sysobjects) < 100
      print "No user-created objects in this
   database" else
    begin
      print "These are the user-created objects"
      select name, type, id
      from sysobjects
      where id > 100
    end
```

The **if...else** condition tests for the presence of user-created objects (all of which have ID numbers that are larger than 100) in a database. Where user tables exist, the **else** clause prints a message and selects their names, types, and ID numbers.

```
4. if (select total_sales
       from titles
       where title_id = “PC9999”) > 100
   select “true”
   else
   select “false”
```

Since the value for total sales for PC9999 in the *titles* is NULL, this query returns “false”. The **else** portion of the query is performed when the **if** portion returns FALSE or NULL. See “Expressions” for more information on truth values and logical expressions.

Comments

- The **if** or **else** conditional affects the performance of only a single SQL statement, unless statements are grouped into a block between the keywords **begin** and **end**, as in Example 3.

  The statement clause could be an **execute** stored procedure command, or any other legal SQL statement or statement block.
- If a **select** statement is used as part of the Boolean expression, it must return a single value.
- **if...else** constructs can be used either in a stored procedure (where they are often used to test for the existence of some parameter) or in *ad hoc* queries as in Examples 1 and 2.
- **if** tests can be nested, either within another **if** or following an **else**.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

See Also

Commands	**begin...end, create procedure**
Topics	Control-of-Flow Language, Expressions, Subqueries

insert

Function

Adds new rows to a table or view.

Syntax

```
insert [into] [database.[owner.]]{table_name|view_name}
   [(column_list)]
   {values (expression [, expression]...)
       |select_statement }
```

Keywords and Options

into – is optional.

column_list – is a list of one or more columns to which data is to be added. Enclose the list in parentheses. The columns can be listed in any order, but the incoming data (whether in a **values** clause or a **select** clause) must be in the same order. If a column has the IDENTITY property, you can substitute the **syb_identity** keyword for the actual column name.

The column list is necessary when some, but not all, of the columns in the table are to receive data. If no column list is given, the **insert** is assumed to affect all of the columns in the receiving table (in **create table** order).

For more information, see ‘‘The Insert Column List’’.

values – is a keyword that introduces a list of expressions.

expression – specifies constant expressions, variables, parameters or null values for the indicated columns. The values list must be enclosed in parentheses and must match the explicit or implicit column list. Enclose character and datetime constants in single or double quotes. See “Datatypes” for more information about data entry rules.

select_statement – is a standard **select** statement used to retrieve the values to be inserted.

Examples

```
1. insert titles
   values("BU2222", "Faster!", "business", "1389",
       null, null, null, "ok", "06/17/87", 0)

2. insert titles
   (title_id, title, type, pub_id, notes, pubdate,
       contract)
   values ('BU1237', 'Get Going!', 'business',
       '1389', 'great', '06/18/86', 1)

3. insert newauthors
     select *
     from authors
     where city = "San Francisco"

4. insert test
     select *
     from test
     where city = "San Francisco"
```

Comments

- **insert** only adds new rows: use **update** to modify column values in a row you've already inserted.

The Insert Column List

- The column list determines the order in which values are entered. For example, suppose you have a table called *newpublishers* that is identical in structure and content to the *publishers* table in *pubs*2. In the example below, the columns in the column list of the *newpublishers* table match the columns of the select list in the *publishers* table.

  ```
  insert newpublishers (pub_id, pub_name)
  select pub_id, pub_name
      from publishers
      where pub_name="New Age Data"
  ```

  The *pub_id* and *pub_name* for "New Age Data" are stored in the *pub_id* and *pub_name* columns of *newpublishers*.

  In the next example, the order of the columns in the column list of the *newpublishers* table does not match the order of the columns of the select list of the *publishers* table.

  ```
  insert newpublishers (pub_id, pub_name)
      select pub_name, pub_id
      from publishers
      where pub_name="New Age Data"
  ```

The result is that the *pub_id* for "New Age Data" is stored in the *pub_name* column of the *newpublishers* table, and the *pub_name* for "New Age Data" is stored in the *pub_id* column of the *newpublishers* table.

- You can leave out items in the column list and **values** list as long as the omitted columns allow null values. See Example 2.

Validating Column Values

- **insert** interacts with the **ignore_dup_key, ignore_dup_row,** and **allow_dup_row** options set with the **create index** command. (See "**create index**" for more information.)
- A rule or **check** constraint can restrict the domain of legal values that can be entered into a column. Rules are created with **create rule** and bound with the system procedure **sp_bindrule. check** constraints are declared with the **create table** statement.
- A default can supply a value if a user does not explicitly enter one. Defaults are created with **create default** and bound with the system procedure **sp_bindefault**, or they are declared with the **create table** statement.
- If an **insert** statement violates domain or integrity rules (see "**create rule**" and "**create trigger**"), or if it is the wrong datatype (see "**create table**" and "Datatypes"), the statement fails and SQL Server displays an error message.

Treatment of Blanks

- Inserting an empty string ("") into a variable character type or *text* column inserts a single space. *char* columns are padded to the defined length.
- All trailing spaces are removed from data inserted into *varchar* columns, except in the case of a string which contains only spaces. Strings which contain only spaces are truncated to a single space. Strings longer than the specified length of a *char*, *nchar*, *varchar*, or *nvarchar* column are silently truncated unless **string_rtruncation** option is set to **on**.

Inserting into text *and* image *Columns*

- An **insert** of a NULL into a *text* or *image* column does not create a valid text pointer, nor does it preallocate 2K bytes per value as would otherwise occur. Use **update** to get a valid text pointer for that column.

Insert Triggers

- You can define a trigger that takes a specified action when an **insert** command is issued on a specified table.

Inserting Rows Selected from Another Table

- You can **select** rows from a table and **insert** them into the same table in a single statement. See Example 4.
- To insert data with **select** from a table that has null values in some fields into a table which does not allow null values, you must provide a substitute value for any NULL entries in the original table. For example, to insert data into an *advances* table that won't allow null values, this example substitutes "0" for the NULL fields:

  ```
  insert advances
  select pub_id, isnull(advance, 0) from titles
  ```

  Without the **isnull** function, this command would insert all the rows with non-null values into *advances*, and produce error messages for all the rows where the *advance* column in *titles* contained NULL.

  If this kind of substitution cannot be made for your data, it is not possible to insert the data containing null values into the columns with the NOT NULL specification.

  Two tables may be identically structured, but differ in whether null values are permitted in some fields. You can use **sp_help** to see the null types of the columns in your table.

Transactions and insert

- When you set chained transaction mode, SQL Server implicitly begins a transaction with the **insert** statement if no transaction is currently active. To complete any inserts, you must **commit** the transaction, or you can **rollback** the changes. For example:

  ```
  insert stores (stor_id, stor_name, city, state)
      values ('9999', 'Books-R-Us', 'Fremont', 'AZ')
  if exists (select t1.city, t2.city
      from stores t1, stores t2
      where t1.city = t2.city
      and t1.state = t2.state
      and t1.stor_id < t2.stor_id)
          rollback transaction
  else
          commit transaction
  ```

In chained transaction mode, this batch begins a transaction and inserts a new row into the *stores* table. If it inserts a row containing the same city and state information as another store in the table, it rolls back the changes to *stores* and ends the transaction. Otherwise, it commits the insertions and ends the transaction. For more information about the chained mode, see "Transactions".

Inserting Values Into IDENTITY Columns

- When inserting a row into a table, do not include the name of the IDENTITY column in the column list, or its value in the values list. If the table consists of only one column, an IDENTITY column, omit the column list and leave the values list empty as follows:

  ```
  insert id_table values()
  ```

- The first time you insert a row into a table, SQL Server assigns the IDENTITY column a value of 1. Each new row gets a column value one higher than the last. This value takes precedence over any defaults declared for the column in the **create table** or **alter table** statement or bound to the column with the **sp_bindefault** system procedure.

  Server failures can create gaps in IDENTITY column values. The maximum size of the gap depends on the setting of the *identity burning set* configuration variable. Gaps can also result from manual insertion of data into the IDENTITY column, deletion of rows, and transaction rollbacks.

- Only the table owner, Database Owner, or System Administrator can explicitly insert a value into an IDENTITY column after setting **identity_insert on** for the column's base table. A user can turn the **identity_insert** option on for one table in a database at a time. When **identity_insert** is turned on, each **insert** statement must include a column list and must specify an explicit value for the IDENTITY column.

  Inserting a value into the IDENTITY column allows you to specify a "seed" value for the column or to restore a row that was deleted in error. Unless you have created a unique index on the IDENTITY column, SQL Server does not verify the uniqueness of the value; you can insert any positive integer.

- The maximum value that can be inserted into an IDENTITY column is $10^{\text{precision}} - 1$. Once an IDENTITY column reaches this value, all further **insert** statements return an error that aborts the current transaction.

When this happens, use the **create table** statement to create a new table identical to the old, but with a larger precision for the IDENTITY column. Once you have created the new table, use either the **insert** statement or the **bcp** utility to copy the data from the old table to the new.

- Use the *@@identity* global variable to retrieve the last value that was inserted into an IDENTITY column. If the last **insert** or **select into** statement affected a table with no IDENTITY column, *@@identity* returns the value 0.
- An IDENTITY column selected into a result table observes the following rules with regard to inheritance of the IDENTITY property:
  - If an IDENTITY column is selected more than once, it is defined as NOT NULL in the new table. It does not inherit the IDENTITY property.
  - If an IDENTITY column is selected as part of an expression, the resulting column does not inherit the IDENTITY property. It is created as NULL if any column in the expression allows nulls, and NOT NULL otherwise.
  - If the select statement contains a **group by** clause or aggregate function, the resulting column does not inherit the IDENTITY property. Columns that include an aggregate of the IDENTITY column are created NULL; others, NOT NULL.
  - An IDENTITY column that is selected into a table with a union or join does not retain the IDENTITY property. If the table contains the union of the IDENTITY column and a NULL column, the new column is defined as NULL. Otherwise, it is defined as NOT NULL.
- To insert an explicit value into an IDENTITY column, the table owner, Database Owner, or System Administrator must set **identity_insert on** for the column's base table, not for the view through which it is being inserted.

Inserting Data through Views

- If a view is created **with check option**, each row that is inserted through the view must meet the selection criteria of the view.

For example, the *stores_cal* view includes all rows of the *stores* table for which *state* has a value of "CA":

```
create view stores_cal
as select * from stores
where state = "CA"
with check option
```

The **with check option** clause checks each **insert** statement against the view's selection criteria. Rows for which *state* has a value other than "CA" are rejected.

- If a view is created **with check option**, all views derived from the "base" view must satisfy the view's selection criteria. Each new row inserted through a derived view must be visible through the base view.

  Consider the view *stores_cal30*, which is derived from *stores_cal*. The new view includes information about stores in California with payment terms of "Net 30":

  ```
  create view stores_cal30
  as select * from stores_cal
  where payterms = "Net 30"
  ```

  Because *stores_cal* was created **with check option**, all rows inserted or updated through *stores_cal30* must be visible through *stores_cal*. Any row with a *state* value other than "CA" is rejected.

  Notice that *stores_cal30* does not have a **with check option** clause of its own. This means that it is possible to insert or update a row with a *payterms* value other than "Net 30" through *stores_cal30*. The following **update** statement would be successful, even though the row would no longer be visible through *stores_cal30*:

  ```
  update stores_cal30
  set payterms = "Net 60"
  where stor_id = "7067"
  ```

- **insert** statements are not allowed on join views created **with check option.**
- If you insert or update a row through a join view, all affected columns must belong to the same base table.

Standards and Compliance

Standard	Compliance Level	Comments
SQL89	Compliant	The following are Transact-SQL extensions: • A **union** operator in the **select** portion of an **insert** statement • Qualification of a table or column name by a database name • Insertion through a view that contains a join (Note: this is not detected by the FIPS flaggger.)
SQL92	Entry level compliant	

Permissions

insert permission defaults to the table or view owner, who can transfer it to other users.

insert permission for a table's IDENTITY column is limited to the table owner, Database Owner, and System Administrator.

See Also

Commands	**create default, create index, create rule, create table, create trigger, delete, select, update**
Topics	Datatypes
System procedures	**sp_bindefault, sp_bindrule, sp_unbindefault, sp_unbindrule**
Utility programs	**bcp**

kill

Function

Kills a process.

Syntax

kill *spid*

Keywords and Options

spid – is the identification number of the process you want to kill. *spid* must be a constant; it cannot be passed as a parameter to a stored procedure or used as a local variable. Use **sp_who** to see a list of processes and other information.

Examples

```
kill 1378
```

Comments

- To get a report on the current processes, execute the system procedure **sp_who**. Here's a typical report:

```
spid status    loginame hostname blk dbname cmd
---- --------  -------- -------- --- ------ --------------
 1 recv sleep  bird     jazzy    0   master AWAITING COMMAND
 2 sleeping    NULL              0   master NETWORK HANDLER
 3 sleeping    NULL              0   master MIRROR HANDLER
 4 sleeping    NULL              0   master AUDIT PROCESS
 5 sleeping    NULL              0   master CHECKPOINT SLEEP
 6 recv sleep  rose     petal    0   master AWAITING COMMAND
 7 running     sa       helos    0   master SELECT
 8 send sleep  daisy    chain    0   pubs2  SELECT
 9 alarm sleep lily     pond     0   master WAITFOR
10 lock sleep  viola    cello    7   pubs2  SELECT
```

The *spid* column contains the process identification numbers used in the Transact-SQL **kill** command. The *blk* column contains the process ID of a blocking process, if there is one. A blocking process (which may have an exclusive lock) is one that is holding resources that another process needs. In this example, process 10 (a **select** on a table) is blocked by process 7 (a **begin transaction** followed by an **insert** on the same table).

The *status* column reports the state of the command. These are the status values and the effects of **sp_who**:

Status	Condition	Effects of kill Command
`recv sleep`	waiting on a network read	immediate
`send sleep`	waiting on a network send	immediate
`alarm sleep`	waiting on an alarm, such as `waitfor delay "10:00"`	immediate
`lock sleep`	waiting on a lock acquisition	immediate
`sleeping`	waiting disk I/O, or some other resource. Probably indicates a process that is running, but doing extensive disk I/O	killed when it "wakes up", usually immediate; a few sleeping processes do not wake up, and require a Server reboot to clear
`runnable`	in the queue of runnable processes	immediate
`running`	actively running on one on the Server engines	immediate
`infected`	Server has detected serious error condition; extremely rare	**kill** command not recommended. Server reboot probably required to clear process
`background`	a process, such as a threshold procedure, run by SQL Server rather than by a user process	immediate; use **kill** with extreme care. Recommend a careful check of *sysprocesses* before killing a background process
`log suspend`	processes suspended by reaching the last-chance threshold on the log	killed when it "wakes up": 1) when space is freed in the log by a **dump transaction** command or 2) when an SA uses the **lct_admin** function to wake up "log suspend" processes

Table 1-17: Status values reported by sp_who

- To get a report on the current locks and the *spid*s of the processes holding them, use the system procedure **sp_lock**. Here's a typical report:

```
  spid    locktype    table_id         page    dbname
------    ----------- -----------      ------  --------
     1    Sh_intent   16003088         0       master
     1    Ex_page     16003088         761     master
     4    Sh_table    112003430        0       pubs2
     4    Ex_table    240003886        0       pubs2
```

  In this example, process 4 has an exclusive table lock and a shared table lock. Process 1 has an exclusive page lock and an intent lock.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

Permissions

kill permission defaults to System Administrators, and is not transferable.

See Also

Commands	**shutdown**
System procedures	**sp_lock, sp_who**

load database

Function

Loads a backup copy of a user database, including its transaction log, that was created with **dump database**. The **listonly** and **headeronly** options display information about the dump files without loading them. Dumps and loads are performed through a Backup Server.

Syntax

```
load database database_name
   from stripe_device [at backup_server_name ]
       [density = density_value,
        blocksize = number_bytes,
        dumpvolume = volume_name,
        file = file_name]
   [stripe on stripe_device [at backup_server_name ]
       [density = density_value,
        blocksize = number_bytes,
        dumpvolume = volume_name,
        file = file_name]
   [stripe on stripe_device [at backup_server_name ]
       [density = density_value,
        blocksize = number_bytes,
        dumpvolume = volume_name,
        file = file_name]...]
   [with {
       density = density_value,
       blocksize = number_bytes,
       dumpvolume = volume_name,
       file = file_name,
       [dismount | nodismount],
       [nounload | unload],
       listonly [= full],
       headeronly,
       notify = {client | operator_console}
       }]
```

Keywords and Options

database_name – is the name of the database that will receive the backup copy. It can be either a newly created database or an existing database. Loading dumped data to an existing database overwrites all existing data. The receiving database must be at least as large as the dumped database. The database can be

specified as a literal, a local variable, or a stored procedure parameter.

from *stripe_device* – is the device from which data is being loaded. See "Specifying Dump Devices" for information about what form to use when specifying a dump device. See the *System Administration Guide Supplement* for a list of supported dump devices.

at *backup_server_name* – is the name of a remote Backup Server running on the machine to which the dump device is attached. For platforms that use *interfaces* files, the *backup_server_name* must appear in the *interfaces* file.

density = *density_value* – this option is ignored.

blocksize = *number_bytes* – overrides the default blocksize for a dump device. Do not specify a blocksize on OpenVMS systems. If you specify a blocksize on UNIX systems, it should be identical to that used to make the dump.

dumpvolume = *volume_name* – is the volume name field of the ANSI tape label. **load database** checks this label when the tape is opened and generates an error message if the wrong volume is loaded.

stripe on *stripe_device* – is an additional dump device. You can use up to 32 devices, including the device named in the **to** *stripe_device* clause. The Backup Server loads data from all devices concurrently, reducing the time and the number of volume changes required. See "Specifying Dump Devices" for information about how to specify a dump device.

dismount | **nodismount** – **on platforms that support logical dismount** (such as OpenVMS), determines whether tapes remain mounted. By default, all tapes used for a load are dismounted when the load completes. Use **nodismount** to keep tapes available for additional loads or dumps.

nounload | **unload** – determines whether tapes rewind after the load completes. By default, tapes do not rewind, allowing you to make additional loads from the same tape volume. Specify **unload** for the last dump file to be loaded from a multi-dump volume. This rewinds and unloads the tape when the load completes.

file = ***file_name*** – is the name of a particular database dump on the tape volume. If you did not record the dump file names at the time you made the dump, use **listonly** to display information about all dump files.

listonly [**= full**] – displays information about all dump files on a tape volume, but **does not load the database**. **listonly** identifies the database and device, the date and time the dump was made, and the date and time it can be overwritten. **listonly = full** provides additional details about the dump. Both reports are sorted by ANSI tape label.

After listing the files on a volume, the Backup Server sends a volume change request. The operator can either mount another tape volume or terminate the list operation for all dump devices.

◆ ***WARNING!***

Do not use *load database with listonly* on quarter-inch cartridge tape.

headeronly – displays header information for a single dump file, but **does not load the database**. **headeronly** displays information about the first file on the tape unless you use the **file = *file_name*** option to specify another file name. The dump header indicates whether the file contains a database or transaction log dump, the database ID, the file name, the date the dump was made, the character set, sort order, page count and next object ID.

notify = {**client** | **operator_console**} – overrides the default message destination.

- On operating systems (such as OpenVMS) that offer an operator terminal feature, volume change messages are always sent to the operator terminal on the machine on which the Backup Server is running. Use **client** to route other Backup Server messages to the terminal session that initiated the **dump database**.
- On operating systems (such as UNIX) that do not offer an operator terminal feature, messages are sent to the client that initiated the **dump database**. Use **operator_console** to route messages to the terminal on which the Backup Server is running.

Examples

1. For UNIX:

```
load database pubs2
  from "/dev/nrmt0"
```

   For OpenVMS:

```
load database pubs2
  from "MTA0:"
```

   Reloads the database *pubs2* from a tape device.

2. For UNIX:

```
load database pubs2
    from "/dev/nrmt4" at REMOTE_BKP_SERVER
    stripe on "/dev/nrmt5" at REMOTE_BKP_SERVER
    stripe on "/dev/nrmt0" at REMOTE_BKP_SERVER
```

   For OpenVMS:

```
load database pubs2
    from "MTA0:" at REMOTE_BKP_SERVER
    stripe on "MTA1:" at REMOTE_BKP_SERVER
    stripe on "MTA2:" at REMOTE_BKP_SERVER
```

   Loads the *pubs2* database, using the Backup Server *REMOTE_BKP_SERVER*. This command names 3 devices.

Comments

Commands Used to Restore Databases from Dumps

Table 1-18 describes the commands and system procedures used to restore databases from backups:

Use This Command	To
create database for load	Create a database for the purpose of loading a dump
load database	Restore a database from a dump
load transaction	Apply recent transactions to a database
load { database | transaction } with headeronly, listonly	Identify the dump files on a tape
sp_volchanged	Respond to the Backup Server's volume change messages

Table 1-18: Commands used to restore databases from dumps

load database *Restrictions*

- You cannot load a dump that was made on a different platform or with an earlier version of SQL Server.
- If a database has cross-database referential integrity constraints, the *sysreferences* system table stores the **name**—not the ID number—of the external database. SQL Server cannot guarantee referential integrity if you use **load database** to change the database name or to load it onto a different server.

◆ ***WARNING!***

Before dumping a database in order to load it with a different name or move it to another SQL Server, use alter table to drop all external referential integrity constraints.

- To recover a database, first load the most recent database dump, and then load all the transaction log dumps made since the last database dump, **in order**. SQL Server checks timestamp on each dump to make sure that it is being loaded to the correct database, and in the correct sequence.
- **load database** overwrites any existing data in the database.
- The receiving database must be as large as, or larger than, the database to be loaded. If the receiving database is too small, SQL Server displays an error message that gives the required size.
- You cannot load from the "null device" (on UNIX, */dev/null*; on OpenVMS, any device name beginning with NL).
- You cannot use the **load database** command in a user-defined transaction.
- Each time you add or remove a cross-database constraint, or drop a table that contains a cross-database constraint, dump **both** of the affected databases.

◆ ***WARNING!***

Loading earlier dumps of these databases could cause database corruption.

Locking Users Out During Loads

- While you are loading a database, it cannot be in use. If database commands are issued while a **load database** or **load transaction** command is running, users get a message telling them that the database is in the process of being loaded.
- If you need to load transaction log dumps as well as the database dump, you should set the following **sp_dboption** options to TRUE before running **load database**: **no chkpt on recovery, dbo use only** and **read only.** This ensures that no user can change the database between loads: transactions that write to the log make it impossible to load the next log dump. Reset these options to FALSE after the last **load.**

Specifying Dump Devices

- You can specify the dump device as a literal, a local variable, or a parameter to a stored procedure.
- You can specify a local device as:
  - A logical device name from the *sysdevices* system table
  - An absolute pathname
  - A relative pathname

  The Backup Server resolves relative pathnames using SQL Server's current working directory.
- When loading across the network, you must specify the absolute pathname of the dump device. The pathname must be valid on the machine on which the Backup Server is running. If the name includes any characters except letters, numbers or the underscore (_), you must enclose it in quotes.
- Ownership and permissions problems on the dump device may interfere with use of load commands.
- You can run more than one load (or dump) at the same time as long as they use different physical devices.

Backup Servers

- You must have a Backup Server running on the same machine as your SQL Server. (On OpenVMS systems, the Backup Server can be running in the same cluster as the SQL Server, as long as all database devices are visible to both.) The Backup Server must be listed in the *master..sysservers* table. This entry is created during installation or upgrade, and should not be deleted.

- If your backup devices are located on another machine, so that you load across a network, you must also have a Backup Server installed on the remote machine.

Volume Names

- Dump volumes are labeled according to the ANSI tape-labeling standard. The label includes the logical volume number and the position of the device within the stripe set.
- During loads, Backup Server uses the tape label to verify that volumes are mounted in the correct order. This allows you to load from a smaller number of devices than you used at dump time.

➤ ***Note***

When dumping and loading across the network, you must specify the same number of stripe devices for each operation.

Changing Dump Volumes

- If the Backup Server detects a problem with the currently mounted volume, it requests a volume change by sending messages to either the client or its operator console. After mounting another volume, the operator notifies the Backup Server by executing the **sp_volchanged** system procedure on any SQL Server that can communicate with the Backup Server.
- On OpenVMS systems, the operating system requests a volume change when the specified drive is off-line. After mounting another volume, the operator uses the REPLY command to reply to these messages.

Restoring the System Databases

- Because the *master*, *model*, and *sybsystemprocs* databases do not store their transaction logs on a separate segment, you cannot use **dump transaction** to make a copy of the transaction log.
- See the *System Administration Guide* for step-by-step instructions on restoring the system databases from dumps.

Disk Mirroring

- At the beginning of a load, SQL Server passes Backup Server the primary device name of each logical database and log device. If

the primary device has been unmirrored, SQL Server passes the name of the secondary device instead. If any named device fails before Backup Server completes its data transfer, SQL Server aborts the load.

- If you attempt to unmirror any named device while a **load database** is in progress, SQL Server displays a message. The user executing the **disk unmirror** command can abort the load or defer the **disk unmirror** until after the load completes.
- The Backup Server loads the data onto the primary device and then **load database** copies it to the secondary device. **load database** takes longer to complete if any database device is mirrored.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

Permissions

Only a System Administrator, Database Owner, or user with the Operator role can execute **load database**.

See Also

Commands	**dbcc, dump database, dump transaction, load transaction**
System procedures	**sp_helpdevice, sp_volchanged**

load transaction

Function

Loads a backup copy of the transaction log that was created with the **dump transaction** command. The **listonly** and **headeronly** options display information about the dump files without loading them. Dumps and loads are performed through a Backup Server.

Syntax

```
load tran[saction] database_name
   from stripe_device [at backup_server_name]
       [density = density_value,
        blocksize = number_bytes,
        dumpvolume = volume_name,
        file = file_name]
   [stripe on stripe_device [at backup_server_name]
       [density = density_value,
        blocksize = number_bytes,
        dumpvolume = volume_name,
        file = file_name]
   [stripe on stripe_device [at backup_server_name]
       [density = density_value,
        blocksize = number_bytes,
        dumpvolume = volume_name,
        file = file_name]...]
   [with {
       density = density_value,
       blocksize = number_bytes,
       dumpvolume = volume_name,
       file = file_name,
       [dismount | nodismount],
       [nounload | unload],
       listonly [= full],
       headeronly,
       notify = {client | operator_console}
   }]
```

Keywords and Options

database_name – is the name of the database to receive data from a dumped backup copy. It can be either a newly created database with no data or an existing database. Loading dumped data to an existing database replaces existing data with the loaded data. The log segment of the receiving database must be at least as large as the log segment of the dumped database. The database can be

specified as a literal, a local variable, or a parameter of a stored procedure.

from *stripe_device* – is the name of the dump device from which you are loading the transaction log. See "Specifying Dump Devices" for information about what form to use when specifying a dump device. See the *System Administration Guide Supplement* for a list of supported dump devices.

at *backup_server_name* – is the name of a remote Backup Server running on the machine to which the dump device is attached. For platforms that use interfaces files, the *backup_server_name* must appear in the interfaces file.

density = *density_value* – overrides the default density for a tape device. **This option is ignored.**

blocksize = *number_bytes* – overrides the default blocksize for a dump device. Do not specify a blocksize on OpenVMS systems. If you specify a blocksize on UNIX systems, it should be identical to that used to make the dump.

dumpvolume = *volume_name* – is the volume name field of the ANSI tape label. **load database** checks this label when the tape is opened and generates an error message if the wrong volume is loaded.

stripe on *stripe_device* – is an additional dump device. You can use up to 32 devices, including the device named in the **to** *stripe_device* clause. The Backup Server loads data from all devices concurrently, reducing the time and the number of volume changes required. See "Specifying Dump Devices" for information about how to specify a dump device.

dismount | **nodismount** – **on platforms that support logical dismount** (such as OpenVMS), determines whether tapes remain mounted. By default, all tapes used for a load are dismounted when the load completes. Use **nodismount** to keep tapes available for additional loads or dumps.

nounload | **unload** – determines whether tapes rewind after the load completes. By default, tapes do not rewind, allowing you to make additional loads from the same tape volume. Specify **unload** for the last dump file to be loaded from a multi-dump volume. This rewinds and unloads the tape when the load completes.

file = ***file_name*** – is the name of a particular database dump on the tape volume. If you did not record the dump file names at the time you

made the dump, use **listonly** to display information about all dump files.

listonly [**= full**] – displays information about all dump files on a tape volume, but **does not load the transaction log**. **listonly** identifies the database and device, the date and time the dump was made, and the date and time it can be overwritten. **listonly = full** provides additional details about the dump. Both reports are sorted by ANSI tape label.

After listing the files on a volume, the Backup Server sends a volume change request. The operator can either mount another tape volume or terminate the list operation for all dump devices

◆ ***WARNING!***

Do not use *load transaction with listonly* on quarter-inch cartridge tape.

headeronly – displays header information for a single dump file, but **does not load the database**. **headeronly** displays information about the first file on the tape unless you use the **file = *file_name*** option to specify another file name. The dump header indicates whether the file contains a database or transaction log dump, the database ID, the file name, the date the dump was made, the character set, sort order, page count and next object ID, the checkpoint location in the log, the location of the oldest **begin transaction** record and old and new sequence dates.

notify = {**client** | **operator_console**} – overrides the default message destination.

- On operating systems (such as OpenVMS) that offer an operator terminal feature, volume change messages are always sent to the operator terminal on the machine on which the Backup Server is running. Use **client** to route other Backup Server messages to the terminal session that initiated the **dump database**.
- On operating systems (such as UNIX) that do not offer an operator terminal feature, messages are sent to the client that initiated the **dump database**. Use **operator_console** to route messages to the terminal on which the Backup Server is running.

Examples

1. For UNIX:

```
load transaction pubs2
  from "/dev/nrmt0"
```

   For OpenVMS:

```
load transaction pubs2
  from "MTA0:"
```

   Loads the transaction log for the database *pubs*2 tape.

2. For UNIX:

```
load database pubs2
    from "/dev/nrmt4" at REMOTE_BKP_SERVER
    stripe on "/dev/nrmt5" at REMOTE_BKP_SERVER
    stripe on "/dev/nrmt0" at REMOTE_BKP_SERVER
```

   For OpenVMS:

```
load database pubs2
    from "MTA0:" at REMOTE_BKP_SERVER
    stripe on "MTA1:" at REMOTE_BKP_SERVER
    stripe on "MTA2:" at REMOTE_BKP_SERVER
```

   Loads the transaction log for the *pubs2* database, using the Backup Server *REMOTE_BKP_SERVER*.

Comments

Commands Used to Restore Databases from Dumps

Table 1-19 describes the commands and system procedures used to restore databases from backups:

Use This Command	To
create database for load	Create a database for the purpose of loading a dump
load database	Restore a database from a dump
load transaction	Apply recent transactions to a database
load { database | transaction } with headeronly, listonly	Identify the dump files on a tape
sp_volchanged	Respond to the Backup Server's volume change messages

Table 1-19: Commands used to restore databases from dumps

load transaction *Restrictions*

- You cannot load a dump that was made on a different platform or with an earlier version of SQL Server.
- You cannot load from the "null device" (on UNIX, */dev/null*; on OpenVMS, any device name beginning with NL).
- You cannot use the **load transaction** command in a user-defined transaction.
- To recover a database, first load the most recent database dump, and then load all the transaction log dumps made since the last database dump, **in order**. SQL Server checks timestamps on the dumped transaction log to make sure that it is being loaded to the correct database, and in the correct sequence.
- For more information on backup and recovery of SQL Server databases, see the *System Administration Guide*.
- Each time you add or remove a cross-database constraint, or drop a table that contains a cross-database constraint, dump **both** of the affected databases.

◆ ***WARNING!***

Loading earlier dumps of these databases could cause database corruption.

Locking Users out During Loads

- While you are loading a database, it cannot be in use. If database commands are issued while a **load database** or **load transaction** command is running, users get a message telling them that the database is in the process of being loaded.
- Set the following **sp_dboption** options to TRUE before running **load database**: **no chkpt on recovery, dbo use only,** and **read only.** This ensures that no user can change the database between loads: transactions that write to the log make it impossible to perform the next load. Set these options to FALSE after the last **load transaction**.

Specifying Dump Devices

- You can specify the dump device as a literal, a local variable, or a parameter to a stored procedure.

- When loading from a local device, you can specify the dump device as:
  - An absolute pathname
  - A relative pathname
  - A logical device name from the *sysdevices* system table

  The Backup Server resolves relative pathnames using SQL Server's current working directory.
- When loading across the network, you must specify the absolute pathname of the dump device. (You cannot use a relative pathname or a logical device name from the *sysdevices* system table.) The pathname must be valid on the machine on which the Backup Server is running. If the name includes any characters except letters, numbers or the underscore (_), you must enclose it in quotes.
- Ownership and permissions problems on the dump device may interfere with use of load commands. The **sp_addumpdevice** procedure adds the device to the system tables, but does not guarantee that you can load from that device or create a file as a dump device.
- You can run more than one load (or dump) at the same time as long as they use different physical devices.

Backup Servers

- You must have a Backup Server running on the same machine as your SQL Server. (On OpenVMS systems, the Backup Server can be running in the same cluster as the SQL Server, as long as all database devices are visible to both.) The Backup Server must be listed in the *master..sysservers* table. This entry is created during installation or upgrade, and should not be deleted.
- If your backup devices are located on another machine, so that you load across a network, you must also have a Backup Server installed on the remote machine.

Volume Names

- Dump volumes are labeled according to the ANSI tape-labeling standard. The label includes the logical volume number and the position of the device within the stripe set.

- During loads, Backup Server uses the tape label to verify that volumes are mounted in the correct order. This allows you to load from a smaller number of devices than you used at dump time.

➤ ***Note***

When dumping and loading across the network, you must specify the same number of stripe devices for each operation.

Changing Dump Volumes

- If the Backup Server detects a problem with the currently mounted volume, it requests a volume change by sending messages to either the client or its operator console. After mounting another volume, the operator notifies the Backup Server by executing the **sp_volchanged** system procedure on any SQL Server that can communicate with the Backup Server.
- On OpenVMS systems, the operating system requests a volume change when the specified drive is off-line. After mounting another volume, the operator uses the REPLY command to reply to these messages.

Restoring the System Databases

- Because the *master*, *model*, and *sybsystemprocs* databases do not store their transaction logs on a separate segment, you cannot use **dump transaction** to make a copy of the transaction log.
- See the *System Administration Guide* for step-by-step instructions on restoring the system databases from dumps.

Disk Mirroring

- At the beginning of a load, SQL Server passes the Backup Server the primary device name of each logical database device and each logical log device. If the primary device has been unmirrored, SQL Server passes the name of the secondary device instead. If any named device fails before the Backup Server completes its data transfer, SQL Server aborts the load.
- If you attempt to unmirror any of the named devices while a **load transaction** is in progress, SQL Server displays a message. The user executing the **disk unmirror** command can abort the load or defer the **disk unmirror** until after the load completes.

- The Backup Server loads the data onto the primary device and then **load transaction** copies it to the secondary device. **load transaction** takes longer to complete if any database device is mirrored.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

Permissions

load transaction permission defaults to the Database Owner and Operators, and is not transferable.

See Also

Commands	**disk unmirror, dump database, dump transaction, load database**
System procedures	**sp_helpdevice, sp_volchanged**

open

Function

Opens a cursor for processing.

Syntax

open ***cursor_name***

Parameters

cursor_name – the name of the cursor to open

Examples

```
open authors_crsr
```

Opens the cursor named *authors_crsr*.

Comments

- SQL Server returns an error message if the cursor is already open or if the cursor has not been created with the **declare cursor** statement.
- You must first open a cursor to use the **fetch**, **update**, and **delete** statements.
- Opening the cursor causes SQL Server to evaluate the **select** statement that defines the cursor (specified in the **declare cursor** statement) and makes the cursor result set available for processing.
- When the cursor is first opened, it is positioned before the first row of the cursor result set.
- When you set the chained transaction mode, SQL Server implicitly begins a transaction with the **open** statement if no transaction is currently active.

Permissions

open permission defaults to all users.

Standards and Compliance

Standard	Compliance Level
SQL89	Compliant
SQL92	Entry level compliant

See Also

Commands	**close, declare cursor, fetch**
Topics	Cursors, Transactions

order by Clause

Function

Returns query results in the specified column(s) in sorted order.

Syntax

[*Start of* **select** *statement*]

```
[order by {[table_name.| view_name.]column_name
        | select_list_number | expression} [asc | desc]
   [,{[table_name.| view_name.] column_name
        select_list_number|expression} [asc |desc]]...]
```

[*End of* **select** *statement*]

Keywords and Options

order by – sorts the results by columns. In Transact-SQL, you can **order by** items that do not appear in the select list. You can sort by a column heading, a column name, an expression, an alias name (if it was specified in the select list), or a number representing the position of the item in the select list (the *select_list_number*).

asc – sorts the results in ascending order. If you do not specify **asc** or **desc**, **asc** is assumed.

desc – sorts the results in descending order.

Examples

```
1. select title, type, price
   from titles
   where price > $9.99
   order by title

2. select type, price, advance
   from titles
   order by type desc
   compute avg(price), avg(advance) by type

3. select title_id, advance/total_sales
   from titles
   order by advance/total_sales

4. select title as BookName, type as Type
   from titles
   order by Type
```

Comments

Restrictions

- Use **order by** to display your query results in a meaningful order. Without an **order by** clause, you cannot control the order in which results are returned by SQL Server.
- The maximum number of columns allowed in an **order by** clause is 16.
- **order by** cannot be used on *text* or *image* datatype columns.
- Subqueries and view definitions cannot include an **order by** clause (or a **compute** clause or the keyword **into**). Conversely, you cannot use a subquery, an aggregate, a variable, or a constant expression in an **order by** list.
- You cannot update the result set of a Server or Language type cursor if it contains an **order by** clause in **its select** statement. See "Cursors" for more information about the restrictions applied to updatable cursors.
- If you use **compute by**, you must also use an **order by** clause. The expressions listed after **compute by** must be identical to or a subset of those listed after **order by**, must be in the same left-to-right order, start with the same expression, and not skip any expressions. For example, suppose that the **order by** clause is:

  ```
  order by a, b, c
  ```

  The **compute by** clause can be any (or all) of these:

  ```
  compute by a, b, c
  compute by a, b
  compute by a
  ```

  The keyword **compute** can be used without **by** to generate grand totals, grand counts, etc. In this case, **order by** is optional.

Collating Sequences

- With **order by**, null values come before all others.
- The sort order (collating sequence) on your SQL Server determines how your data is sorted. The sort order choices are: binary, dictionary, case-insensitive, case-insensitive with

preference, and case- and accent-insensitive. Sort orders that are particular to specific national languages may also be provided.

SQL Server's Sort Order	Effects on *order by* Results
Binary order	Sorts all data according to the numeric byte-value of each character in the character set. Binary order sorts all uppercase letters before lowercase letters. Binary sort order is the only option for multibyte character sets.
Dictionary order	Sorts uppercase letters before their lowercase counterparts (case-sensitive). Dictionary order recognizes the various accented forms of a letter and sorts them after the unaccented form.
Dictionary order, case-insensitive	Sorts data in dictionary order but does not recognize case differences. Uppercase letters are equivalent to their lowercase counterparts and will be sorted as described below.
Dictionary order, case-insensitive with preference	Sorts an uppercase letter in the preferred position, before its lowercase version. It does not recognize case difference when performing comparisons (for example, in **where** clauses).
Dictionary order, case- and accent-insensitive	Sorts data in dictionary order but does not recognize case differences; treats accented forms of a letter as equivalent to the associated unaccented letter. It intermingles accented and unaccented letters in sorting results.

Table 1-20: Effect of sort order choices

- The system procedure **sp_helpsort** reports the sort order installed on your server. When two rows have equivalent values in the Server's sort order, the following rules are used to order the rows:
  - The values in the columns named in the **order by** clause are compared.
  - If two rows have equivalent column values, the binary value of the entire rows is compared byte by byte. This comparison is performed on the row in the order in which the columns are stored internally, not the order of the columns as they are named in the query or in the original **create table** clause. (In brief, data is stored with all the fixed-length columns in order, then all the variable length columns in order.)
  - If rows are equal, row IDs are compared.

Given this table:

```
create table sortdemo (lname varchar(20),
                       init char(1) not null)
```

And this data:

```
lname      init
---------- ----
Smith      B
SMITH      C
smith      A
```

You get these results when you order by *lname*:

```
lname      init
---------- ----
smith      A
Smith      B
SMITH      C
```

Since the fixed-length *char* data (the *init* column) is stored first internally, the **order by** sorts these rows based on the binary values "Asmith", "BSmith" and "CSMITH".

If *init* is a *varchar* instead, the *lname* column is stored first internally, then the *init* column, and the comparison takes place on the binary values "SMITHC", "SmithB" and "smithA", and the rows are returned in that order.

Standards and Compliance

Standard	Compliance Level	Comments
SQL89	Compliant	Specifying new column headings in the **order by** clause of a **select** statement when the **union** operator is used is a Transact-SQL extension.
SQL92	Entry level compliant	

See Also

Commands	**compute** Clause, **declare**, **group by** and **having** Clauses, **select**, **where** Clause
Topics	Cursors, Expressions

prepare transaction

Function

Used by DB-Library in a two-phase commit application to see if a server is prepared to commit a transaction.

Syntax

```
prepare transaction
```

Comments

- See the *Open Client DB-Library Reference Manual* for more information.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

See Also

Commands	begin transaction, commit, rollback, save transaction,
Topics	Transactions

print

Function

Prints a user-defined message on the user's screen.

Syntax

```
print
   {format_string | @local_variable | @@global_variable}
        [, arg_list]
```

Keywords and Options

format_string – may be either a variable or a string of characters. The maximum length of *format_string* is 255 bytes.

Format strings can contain up to 20 unique place holders in any order. These placeholders are replaced with the formatted contents of any arguments that follow *format_string* when the text of the message is sent to the client.

To allow reordering of the arguments when format strings are translated to a language with a different grammatical structure, the place holders are numbered. A place holder for an argument appears in this format: "*%nn*!", a percent sign (%), followed by an integer from 1 to 20, followed by an exclamation point (!). The integer represents the argument number in the string in the argument list. "%1!" is the first argument in the original version, "%2!" is the second argument, and so on.

Indicating the position of the argument in this way makes it possible to translate correctly, even when the order in which the arguments appear in the target language is different.

For example, assume the following is an English message:

```
%1! is not allowed in %2!.
```

The German version of this message is:

```
%1! ist in %2! nicht zulassig.
```

The Japanese version of this message is:

%2！の中で %1！は許されません。

In this example, "%1!" in all three languages represents the same argument, and " %2!" also represents a single argument in all three languages. This example shows the reordering of the arguments that is sometimes necessary in the translated form.

@local_variable – must be of type *char*, *nchar*, *varchar*, or *nvarchar*, and must be declared within the batch or procedure in which it is used.

@@global_variable – must be of type *char* or *varchar* or automatically convertible to these types, such as *@@version*. Currently, *@@version* is the only character-type global variable.

arg_list – may be a series of either variables or constants separated by commas. *arg_list* is optional unless a format string containing place holders of the form *%nn!* is provided. In that case the *arg_list* must have at least as many arguments as the highest numbered place holder. An argument can be any datatype except *text* or *image*; it is converted to a character datatype before being included in the final message.

Examples

1.
```
if exists (select postalcode from authors
where postalcode = '94705')
print "Berkeley author"
```

2.
```
declare @msg char(50)
select @msg = "What's up, doc?"
print @msg
```

```
What's up, doc?
```

3.
```
declare @tabname varchar(30)
select @tabname = "titles"

declare @username varchar(30)
select @username = "ezekiel"
print "The table '%1!' is not owned by the user
'%2!'.", @tabname, @username
```

```
The table 'titles' is not owned
by the user 'ezekiel.'
```

Comments

- The maximum output string length of *format_string* plus all arguments after substitution is 512 bytes.
- If you use place holders in a format string, keep this in mind: for each place holder *n* in the string, the place holders 1 through *n*-1 must also exist in the same string (although they do not have to be in numerical order). For example, you cannot have place holders 1 and 3 in a format string without having the place holder

2 in the same string. If you do omit a number in a format string, an error message is generated when **print** is executed.

- The *arg_list* must include an argument for each placeholder in the *format_string*, or the transaction is aborted. It is permissible to have more arguments than placeholders.
- To include a literal percent sign as part of the error message, use two percent signs ("%%") in the *format_string*. If you include a single percent sign ("%") in the *format_string* which is not used as a placeholder, SQL Server returns an error message.
- If an argument evaluates to NULL, it is converted into a zero-length character string. If you do not want zero-length strings in the output, use the **isnull** function (see "System Functions"). For example:

```
declare @arg varchar(30)
select @arg = isnull(col1, "nothing") from
table_a where ...

print "I think we have %1! here", @arg
```

prints:

```
I think we have nothing here.
```

if *@arg* is null.

- User-defined messages can be added to the system table *sysusermessages* for use by any application. Use **sp_addmessage** to add messages to *sysusermessages*; use **sp_getmessage** to retrieve messages for use by **print** and **raiserror.**
- Use **raiserror** instead of **print** if you want to print a user-defined error message and have the error number stored in *@@error*.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

See Also

Commands	**declare**, **raiserror**
Topics	Variables (Local and Global)
System procedures	**sp_addmessage**, **sp_getmessage**

raiserror

Function

Prints a user-defined error message on the user's screen and sets a system flag to record that an error condition has occurred.

Syntax

```
raiserror error_number
   [{format_string | @local_variable}] [, arg_list]
   [extended_value = extended_value [{,
   extended_value = extended_value}...]]
```

Keywords and Options

error_number – is an integer with a value greater than 17,000. If the *error_number* is between 17,000 and 19,999, and *format_string* is missing or empty (""), SQL Server retrieves error message text from the *sysmessages* table in the *master* database. These error messages are used chiefly by system procedures.

If *error_number* is 20,000 or greater and *format_string* is missing or empty, **raiserror** retrieves the message text from the *sysusermessages* table in the database from which the query or stored procedure originates. SQL Server attempts to retrieve messages from either *sysmessages* or *sysusermessages* in the language defined by the current setting of *@@langid*.

format_string – is a string of characters with a maximum length of 255 bytes. Optionally, you can declare *format_string* in a local variable and use that variable with **raiserror** (see *@local_variable*).

raiserror recognizes place holders in the character string that is to be printed out. Format strings may contain up to 20 unique place holders in any order. These placeholders are replaced with the formatted contents of any arguments that follow *format_string* when the text of the message is sent to the client.

To allow reordering of the arguments when format strings are translated to a language with a different grammatical structure, the place holders are numbered. A place holder for an argument appears in this format: *%nn!*, a percent sign (%), followed by an integer from 1 to 20, followed by an exclamation point (!). The integer represents the argument number in the string in the argument list. "%1!" is the first argument in the original version, "%2!" is the second argument, and so on.

Indicating the position of the argument in this way makes it possible to translate correctly even when the order in which the arguments appear in the target language is different from their order in the source language.

For example, assume the following is an English message:

```
%1! is not allowed in %2!.
```

The German version of this message is:

```
%1! ist in %2! nicht zulassig.
```

The Japanese version of this message is:

%2! の中で %1! は許されません。

In this example, "%1!" in all three languages represents the same argument, and "%2!" also represents a single argument in both languages. This example shows the reordering of the arguments that is sometimes necessary in the translated form.

@local_variable – is a local variable containing the *format_string* value. It must be of type *char* or *varchar* and must be declared within the batch or procedure in which it is used.

arg_list – is a series of variables or constants separated by commas. *arg_list* is optional unless a format string containing place holders of the form *%nn!* is provided. An argument can be any datatype except *text* or *image*; it is converted to the *char* datatype before being included in the final string.

If an argument evaluates to NULL, SQL Server converts it into a zero-length *char* string.

extended_value – is a string of characters with a maximum length of 255 bytes. Optionally, you can use a global or local variable in place of an *extended_value*.

When SQL Server detects certain errors, it attempts to return the names of the tables and columns involved with the error message to the Open-Client application. These errors include duplicate key violations, rule violations, and integrity constraint violations. Client applications can use these values (called extended error data) to indicate which columns are involved in the error or other information related to error message.

raiserror extends this capability to user-defined error messages. For example, you can specify the extended error data with an error message in a trigger or stored procedure to indicate which table column is involved. When the trigger or stored procedure

detects that error, SQL Server passes the table column value to your client application.

The names of each extended error data value specified for user-defined error messages must be known by the client application to access its information. For more information, see your Open-Client documentation.

Examples

1. ```
   create procedure showtable_sp @tabname varchar(18)
   as
   if not exists (select name from sysobjects
       where name = @tabname)
       begin
           raiserror 99999 "Table %1! not found.",
               @tabname
       end
   else
       begin
           select sysobjects.name, type, crdate, indid
           from sysindexes, sysobjects
           where sysobjects.name = @tabname
           and sysobjects.id = sysindexes.id
       end
   ```

   This stored procedure example returns an error if it does not find the table supplied with the *@tabname* parameter variable.

2. ```
   sp_addmessage 25001,
   "There is already a remote user named '%1!'
   for remote server '%2!'."

   raiserror 25001, jane, myserver
   ```

   This example adds a message to *sysusermessages*, then tests the message with **raiserror**, providing the substitution arguments.

3. ```
   raiserror 20100 "Login must be at least 5
       characters long" "column" = "login",
       "server" = @@servername
   ```

   This example returns the extended error data *column* and *transtate* to a client application to indicate which column was involved and to indicate which server was used.

Comments

- User-defined messages can be generated *ad hoc*, as in the example above, or they can be added to the system table *sysusermessages* for use by any application. Use **sp_addmessage** to add messages to
   ```

sysusermessages; use **sp_getmessage** to retrieve messages for use by **print** and **raiserror.**

- Error numbers for user-defined error messages must be greater than 20,000. The maximum value is 2,147,483,647 (2^{31} -1).
- The severity level of all user-defined error messages is 16. This level indicates that the user has made a a non-fatal mistake.
- The maximum output string length of *format_string* plus all arguments after substitution is 512 bytes.
- If you use place holders in a format string, keep this in mind: for each place holder *n* in the string, the place holders *1* through *n-1* must also exist in the same string (although they do not have to be in numerical order). For example, you cannot have place holders 1 and 3 in a format string without having the place holder 2 in the same string. If you do omit a number in a format string, an error message is generated when **raiserror** is executed.
- If there are too few arguments relative to the number of placeholders in *format_string*, an error message displays and the transaction is aborted. It is permissible to have more arguments than placeholders in *format_string*.
- To include a literal percent sign as part of the error message, use two per cent signs (“%%”) in the *format_string*. If you include a single per cent sign (“%”) in the *format_string* which is not used as a placeholder, SQL Server returns an error message.
- If an argument evaluates to NULL, it is converted into a zero-length *char* string. If you do not want zero-length strings in the output, use the **isnull** function (see the section “System Functions” in this chapter).
- When **raiserror** is executed, the error number is placed in the global variable *@@error*, which stores the error number most recently generated by the system.
- Use **raiserror** instead of **print** if you want an error number stored in *@@error*.
- To include an *arg_list* with **raiserror**, put a comma after *error_number* or *format_string* before the first argument. To include extended error data, separate the first *extended_value* from *error_number*, *format_string*, or *arg_list* using a space (not a comma).

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

See Also

Commands	**declare**, **print**
Topics	Variables (Local and Global)
System procedures	**sp_addmessage**, **sp_getmessage**

readtext

Function

Reads *text* and *image* values, starting from a specified offset and reading a specified number of bytes or characters.

Syntax

```
readtext [[database.]owner.]table_name.column_name
   text_pointer offset size [holdlock]
   [using {bytes | chars | characters}]
```

Keywords and Options

table_name.column_name – the name of the *text* or *image* column must include the table name. The database name and owner name are optional.

text_pointer– a *varbinary(16)* value that stores the pointer to the *text* or *image* data. Use the **textptr** function to determine this value, as shown in the example below. Text and image data is not stored in the same set of linked pages as other table columns. It is stored in a separate set of linked pages. A pointer to the actual location is stored with the data; **textptr** returns this pointer.

offset – specifies the number of bytes or characters to skip before starting to read *text* or *image* data.

size – specifies the number of bytes or characters of data to read.

holdlock – causes the text value to be locked for reads until the end of the transaction. Other users can read the value but they cannot modify it.

using – specifies whether **readtext** interprets the *offset* and *size* parameters as a number of bytes (**bytes**) or as a number of characters (**chars** or **characters** are synonymous). This option has no effect when used with a single-byte character set or with *image* values (**readtext** reads *image* values only on a byte-by-byte basis). If the **using** option is not given, **readtext** interprets the *size* and *offset* arguments as bytes.

Examples

```
create table texttest
(title_id varchar(6), blurb text null,
    pub_id char(4))

insert texttest values ("BU1032",
"The Busy Executive's Database Guide is an
overview of available database systems with
emphasis on common business applications.
Illustrated.", "1389")

declare @val varbinary(16)
select @val = textptr(blurb) from texttest
 where title_id = "BU1032"
readtext texttest.blurb @val 1 5 using chars
```

After creating the table *texttest* and entering values into it, this example selects the second through the sixth character of the *blurb* column.

Comments

- The **textptr** function returns a 16-byte binary string (text pointer) to the *text* or *image* column in the specified row, or to the *text* or *image* column in the last row returned by the query, if more than one row is returned. It is best to declare a local variable to hold the text pointer, and then use the variable with **readtext**.
- The value in the global variable *@@textsize*, which is the limit on the number of bytes of data to be returned, supersedes the size specified for **readtext** if it is less than that size. Use **set textsize** to change the value of *@@textsize*.
- When using bytes as the offset and size, SQL Server may find partial characters at the beginning or end of the *text* data to be returned. If it does and character set conversion is on, the server replaces each partial character with a question mark (?) before returning the text to the client.
- SQL Server has to determine the number of bytes to send to the client in response to a **readtext** command. When the offset and size are in bytes, determining the number of bytes in the returned text is simple. When the offset and size are in characters, the server must take an extra step to calculate the number of bytes being returned to the client. As a result, performance may be slower when using characters as the offset and size. **using characters** is only useful when SQL Server is using a multibyte character set: this option insures that **readtext** will not return partial characters.

- You cannot use **readtext** on *text* and *image* columns in views.
- If you attempt to use **readtext** on *text* values after changing to a multibyte character set and have not run **dbcc fix_text**, the command fails and an error message is generated instructing you to run **dbcc fix_text** on the table.

Standards and Compliance

Standard	Compliance level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

Permissions

readtext requires **select** permission on the table. **readtext** permission is transferred when **select** permission is transferred.

See Also

Commands	**set, writetext**
Functions	text and image Functions
Topics	text and image Datatypes

reconfigure

Function

Sets configuration variables that control various aspects of SQL Server's memory allocation, performance, and options. Used with the system procedure **sp_configure.**

Syntax

```
reconfigure [with override]
```

Keywords and Options

with override – instructs SQL Server to accept the values for system variables supplied by the user (with **sp_configure**). If the user supplies values to **sp_configure** that SQL Server cannot accept, an error message is displayed.

The **with override** clause must always be used when you set **sp_configure**'s **allow updates** variable to "1."

Examples

```
reconfigure
```

The dynamic values in the system table *master..sysconfigures* go into effect immediately. The static values take effect the next time you restart SQL Server.

Comments

- To display a list of the configuration variables along with their current values and the range of permissible values, execute **sp_configure** without specifying any parameters.
- To change a configuration variable, execute **sp_configure** with its parameters *optname* (the name of the configuration variable) and *optvalue* (a legal value). Then issue the **reconfigure** command. Include the **with override** clause if you are turning the **allow updates** variable on.
- SQL Server checks to make sure that the new values in *sysconfigures* are acceptable (SQL Server is able to run) and reasonable (SQL Server will run optimally).

➤ ***Note***

Do not use **reconfigure** to change the default sort order of SQL Server. See the *System Administration Guide* for information about changing sort order.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

Permissions

reconfigure permission defaults to System Administrators, and is not transferable.

See Also

System procedures	**sp_configure**

return

Function

Exits from a batch or procedure unconditionally, optionally providing a return status. Statements following **return** are not executed.

Syntax

```
return [integer_expression]
```

Keywords and Options

integer_expression – is the integer value returned by the procedure. Stored procedures can return an integer value to a calling procedure or application program.

Examples

1.
```
create procedure findrules @nm varchar(30) = null
as
if @nm is null
begin
  print "You must give a user name"
  return
end
else
begin
   select sysobjects.name, sysobjects.id,
   sysobjects.uid
 from sysobjects, master..syslogins
   where master..syslogins.name = @nm
   and sysobjects.uid = master..syslogins.suid
   and sysobjects.type = "R"
end
```

If no user name is given as a parameter, the **return** command causes the procedure to exit after a message has been sent to the user's screen. If a user name is given, the names of the rules created by this user in the current database are retrieved from the appropriate system tables.

2.
```
print "Begin update batch"
update titles
    set price = price + $3
    where title_id = 'BU2075'
update titles
    set price = price + $3
    where title_id = 'BU1111'
if (select avg(price) from titles
        where title_id like 'BU%') > $15
begin
    print "Batch stopped; average price over $15"
    return
end
update titles
    set price = price + $2
    where title_id = 'BU1032'
```

   If the updates cause the average price of business titles to exceed $15, the **return** command terminates the batch before any more updates are performed on *titles*.

3.
```
create proc checkcontract @param varchar(11)
as
declare @status int
if (select contract from titles where title_id =
@param) = 1
   return 1
else
   return 2
```

   This procedure creates two user-defined status codes: a value of 1 is returned if the *contract* column contains a 1; a value of 2 is returned for any other condition (a value of 0 on *contract*, or a *title_id* that did not match a row).

Comments

- The return status value can be used in subsequent statements in the batch or procedure that executed the current procedure, but must be given in the form:

  execute *@retval = procedure_name*

  See "**execute**" for more information.

- SQL Server reserves 0 to indicate a successful return, and negative values in the range -1 to -99 to indicate different reasons for failure. If no user-defined return value is provided, the SQL Server value is used. User-defined return status values must not

conflict with those reserved by SQL Server. Numbers 0 and -1 to -14 are currently in use:

Value	Meaning
0	Procedure executed without error
-1	Missing object
-2	Datatype error
-3	Process was chosen as deadlock victim
-4	Permission error
-5	Syntax error
-6	Miscellaneous user error
-7	Resource error, such as out of space
-8	Non-fatal internal problem
-9	System limit was reached
-10	Fatal internal inconsistency
-11	Fatal internal inconsistency
-12	Table or index is corrupt
-13	Database is corrupt
-14	Hardware error

Table 1-21: SQL Server error return values

Values -15 to -99 are reserved for future SQL Server use.

- If more than one error occurs during execution, the status with the highest absolute value is returned. User-defined return values always take precedence over SQL Server-supplied return values.
- The **return** command can be used at any point where you want to exit from a batch or procedure. Return is immediate and complete: statements after **return** are not executed.
- A stored procedure cannot return a NULL return status. If a procedure attempts to return a null value, for example, using **return** *@status* where *@status* is NULL, a warning message is generated, and a value in the range of 0 to -14 is returned.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

See Also

Commands	**begin...end, execute, if...else, while**

revoke

Function

Revokes permissions from users.

Syntax

To revoke permission to access database objects:

```
revoke [grant option for]
   {all [privileges] | permission_list}
   on { table_name [(column_list)]
        | view_name [(column_list)]
        | stored_procedure_name}
   from {public | name_list | role_name}
   [cascade]
```

To revoke permission to create database objects:

```
revoke {all [privileges] | command_list}
   from {public | name_list | role_name}
```

Keywords and Options

all – when used to revoke permission to access database objects (the first syntax format), **all** specifies that all permissions applicable to the specified object are revoked. All object owners can use **revoke all** with an object name to revoke permissions on their own objects.

Only the System Administrator or the Database Owner can revoke permission to revoke create database objects (the second syntax format). When used by the System Administrator, **revoke all** revokes all **create** permissions (**create database**, **create default**, **create procedure**, **create rule**, **create table**, and **create view**). When the Database Owner uses **revoke all**, SQL Server revokes all **create** permissions except **create database**, and prints an informational message.

permission_list – is a list of permissions to be revoked. If more than one permission is listed, separate them with commas. The

following table illustrates the access permissions that can be granted and revoked on each type of object:

Object	*permission_list* Can Include:
table or view	**select, insert, delete, update, references**
column	**select, update, references** Column names can be specified in either *permission_list* or *column_list* (see example 2).
stored procedure	**execute**

Table 1-22: Object access permissions

Permissions can only be revoked by the user who granted them.

command_list – is a list of object creation permissions to be revoked. If more than one command is listed, separate them with commas. The command list can include **create database**, **create default**, **create procedure**, **create rule**, **create table**, and **create view**. **create database** permission can be revoked only by a System Administrator and only from the *master* database.

table_name – is the name of the table on which you are revoking permissions. The table must be in your current database. Only one object can be listed for each **revoke** statement.

column_list – is a list of columns, separated by commas, to which the privileges apply. If columns are specified, only **select** and **update** permissions can be revoked.

view_name – is the name of the view on which you are revoking permissions. The view must be in your current database. Only one object can be listed for each **revoke** statement.

stored _procedure_name – is the name of the stored procedure on which you are revoking permissions. The stored procedure must be in your current database. Only one object can be listed for each **revoke** statement.

public – is all users. For object access permissions, **public** excludes the object owner. For object creation permissions, **public** excludes the Database Owner (who "owns" object creation permissions within the database).

name_list – is a list of user and / or group names, separated by commas.

role_name – is the name of a SQL Server role. This allows you to revoke permissions from all users who have been granted a specific role. The roles are **sa_role** (System Administrator), **sso_role** (System Security Officer), and **oper_role** (Operator). Roles are granted with **sp_role**.

grant option for – revokes **with grant option** permissions, so that the user(s) specified in *name_list* can no longer grant the specified permissions to other users. If those other users have granted permissions to other users, you must use the **cascade** option to revoke permissions from them as well. The user specified in *name_list* retains permission to access the object, but can no longer grant access to other users. **grant option for** applies only to object access permissions, not to object creation permissions.

cascade – for use with **grant option for**: revokes the specified object access permissions from all users to whom the revokee granted permissions. Applies only to object access permissions, not to object creation permissions. (When you use **revoke** without **grant option for**, permissions granted to other users by the revokee are also revoked: the cascade occurs automatically.)

Examples

1.
```
revoke insert, delete
on titles
from mary, sales
```

   Revokes **insert** and **delete** permissions on the *titles* table from Mary and the "sales" group.

2.
```
revoke update
on titles (price, advance)
from public
```

   or:

```
revoke update (price, advance)
on titles
from public
```

   Two ways to revoke **update** permission on the *price* and *advance* columns of the *titles* table from "public".

3. **revoke create database, create table**
   from mary, john

   Revokes permission to use the **create database** and **create table** commands from Mary and John. Because **create database** permission is being revoked, this command must be executed by a System Administrator from within the *master* database. Mary and John's **create table** permission will be revoked only within the *master* database.

4. **revoke all**
   from mary

   Revokes all object creation permissions from Mary in the current database.

5. **revoke all**
   on titles
   from mary

   Revokes all object access permissions on the *titles* table from Mary.

6. **revoke references**
   on titles (price, advance)
   from tom

   or:

   revoke references (price, advance)
   on titles
   from tom

   Two ways to revoke Tom's permission to create a referential integrity constraint on another table that refers to the *price* and *advance* columns of the *titles* table.

7. **revoke execute on new_sproc**
   from oper_role

   Revokes permission to execute the *new_sproc* stored procedure from all users who have been granted the Operator role.

8. **revoke grant option for**
   insert, update, delete
   on authors
   from john
   cascade

   Revokes John's permission to grant **insert, update**, and **delete** permissions on the *authors* table to other users. Also revokes from other users any such permissions that John has granted.

Comments

- See the **grant** command for a table covering permissions.
- You can only grant or revoke permissions on objects in your current database.
- You can only revoke permissions that were granted by you.
- **grant** and **revoke** commands are order-sensitive. When there is a conflict, the command issued most recently takes effect.
- The word **to** can be substituted for the word **from** in the **revoke** syntax.
- Permissions granted to roles override permissions granted to individual users or groups. Therefore, if you revoke a permission from a user who has been granted a role, and the role has that same permission, the user will retain it. For example, say John has been granted the System Security Officer role, and **sso_role** has been granted permission on the *sales* table. If John's individual permission on *sales* is revoked, he is still able to access *sales* because his role permissions override his individual permissions.
- Revoking a specific permission from "public" or from a group also revokes it from users who were individually granted the permission.
- Database user groups allow you to **grant** or **revoke** permissions to more than one user at a time. A user can only be a member of one group and is always a member of the default group, "public". SQL Server's installation script assigns a set of permissions to "public".

  Groups are created with the system procedure **sp_addgroup** and removed with **sp_dropgroup**. New users can be added to a group with **sp_adduser**. A user's group membership can be changed with **sp_changegroup**. To display the members of a group, use **sp_helpgroup**.
- If you do not specify **grant option for** in a **revoke** statement, **with grant option** permissions are revoked from the user along with the specified object access permissions. In addition, if the user has granted the specified permissions to any other users, all of those permissions are revoked. In other words, the **revoke** automatically cascades.
- **revoke grant option** revokes the user's ability to grant the specified permission to other users, but does not revoke the permission itself from that user. If the user has granted that permission to

others, you must use the **cascade** option or you receive an error message and the **revoke** fails.

For example, say you revoke the **with grant option** from the user Bob on *titles*, with this statement:

```
revoke grant option for all
for select
on titles
from bob
cascade
```

- If Bob has not granted this permission to other users, this command revokes his ability to grant this permission to others, but he retains **select** permission on the *titles* table.
- If Bob has granted this permission to other users, you must use the **cascade** option. If you do not, you receive an error message and the **revoke** fails. **cascade** revokes this **select** permission from all users to whom Bob has granted it, as well as their ability to grant it to others.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

See Also

Commands	**grant, setuser**
Topics	Roles
System procedures	**sp_addgroup, sp_adduser, sp_changedbowner, sp_changegroup, sp_dropgroup, sp_dropuser, sp_helpgroup, sp_helprotect, sp_helpuser, sp_role**

rollback

Function

Rolls a user-defined transaction back to the last savepoint inside the transaction or to the beginning of the transaction.

Syntax

```
rollback {transaction | tran | work}
   [transaction_name | savepoint_name]
```

Keywords and Options

transaction_name – is the name assigned to the transaction. It must conform to the rules for identifiers.

savepoint_name – is the name assigned to the savepoint in the **save transaction** statement. The name must conform to the rules for identifiers.

Comments

- See "Transactions" in Chapter 3 for full information on using transaction management statements and on the effects of **rollback** on stored procedures, triggers and batches.
- **rollback**, without a savepoint name, cancels an entire transaction. All of the transaction's statements or procedures are undone.
- To cancel part of a transaction, use **rollback** with a *savepoint_name*. A savepoint is a marker set by the user within a transaction using the command **save transaction**. All of the statements or procedures between the savepoint and the **rollback** are undone.

  After a transaction is rolled back to a savepoint, it can proceed to completion (executing any SQL statements after that **rollback**) using **commit**, or it can be canceled altogether using **rollback** without a savepoint. There is no limit on the number of savepoints within a transaction.
- If no *savepoint_name* or *transaction_name* is given with the **rollback** command, the transaction is rolled back to the first **begin transaction** in the batch. This also includes transactions started through an implicit **begin transaction** using the chained transaction mode.

  In triggers or stored procedures, **rollback** statements, without transaction or savepoint names, roll back all statements to the

first explicit or implicit **begin transaction** in the batch that called the procedure or fired the trigger.

- The **rollback** command must appear within a transaction. You cannot roll back a transaction after **commit** has been entered.
- When a trigger contains a **rollback** command without a savepoint name, the rollback aborts the entire batch, and any statements in the batch following the rollback will not be executed.
- If no transaction is currently active, the **commit** or **rollback** statement has no effect.
- A remote procedure call (RPC) is executed independently from any transaction in which it is included. In a standard transaction (that is, not using Open Client DB-Library two-phase commit), commands executed via an RPC by a remote server are not rolled back with **rollback**, and do not depend on **commit** to be executed.

Standards and Compliance

Standard	Compliance Level	Comments
SQL89	Compliant	The **rollback transaction** and **rollback tran** forms of the statement and the use of a transaction name are Transact-SQL extensions.
SQL92	Entry level compliant	

Permissions

rollback permission defaults to "public". No permission is required to use it.

See Also

Commands	**begin transaction, commit, create trigger, save transaction,**
Topics	Transactions

rollback trigger

Function

Rolls back the work done in a trigger, including the data modification that caused the trigger to fire, and issues an optional **raiserror** statement.

Syntax

```
rollback trigger
   [with raiserror_statement]
```

Keywords and Options

with *raiserror_statement* – specifies a **raiserror** statement which prints a user-defined error message and sets a system flag to record that an error condition has occurred. This provides the ability to raise an error to the client when the **rollback trigger** is executed, so that the transaction state in the error reflects the rollback. For information about the syntax and rules defining *raiserror_statement*, see **raiserror** in this chapter.

Examples

```
rollback trigger with raiserror 25002
    "title_id does not exist in titles table."
```

Rolls back a trigger and issues the user-defined error message 25002.

Comments

- When the **rollback trigger** is executed, SQL Server aborts the currently executing command and halts execution of the rest of the trigger.
- If the trigger that issues the **rollback trigger** is nested within other triggers, SQL Server rolls back all work done in these triggers up to and including the update which caused the first trigger to fire.
- SQL Server ignores a **rollback trigger** executed outside of a trigger and does not issue a **raiserror** associated with the statement. However, a **rollback trigger** executed outside of a trigger but inside a transaction generates an error which causes SQL Server to roll back the transaction and abort the current statement batch.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

Permissions

rollback trigger permission defaults to "public". No permission is required to use it.

See Also

Commands	**create trigger**, **raiserror**, **rollback**
Topics	Transactions

save transaction

Function

Sets a savepoint within a transaction.

Syntax

```
save transaction savepoint_name
```

Keywords and Options

savepoint_name – is the name assigned to the savepoint. It must conform to the rules for identifiers.

Comments

- See "Transactions" in this chapter for full information on using transaction statements.
- A savepoint is a user-defined marker within a transaction that allows portions of a transaction to be rolled back. The command **rollback** *savepoint_name* rolls back to the indicated savepoint; all of the statements or procedures between the savepoint and the **rollback** are undone.

  Statements preceding the savepoint are not undone—but neither are they committed. After a transaction is rolled back to a savepoint, it can proceed to completion (executing any statements after that **rollback**) using **commit**, or it can be canceled altogether using **rollback** without a savepoint. There is no limit on the number of savepoints within a transaction.
- To cancel an entire transaction, use **rollback** without a transaction name. All of the transaction's statements or procedures are undone, back to the first **begin transaction** in the batch.
- If no *savepoint_name* or *transaction_name* is given with the **rollback** command, all statements back to the first **begin transaction** in a batch are rolled back and the transaction is cancelled.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

Permissions

save transaction permission defaults to "public." No permission is required to use it.

See Also

Commands	**begin transaction**, **commit**, **rollback**
Topics	Transactions

select

Function

Retrieves rows from database objects.

Syntax

```
select [all | distinct] select_list
   [into [[database.]owner.]table_name]
   [from [[database.]owner.]{table_name |view_name}
            [holdlock | noholdlock] [shared]
        [,[[database.]owner.]{table_name |view_name}
            [holdlock | noholdlock] [shared]]... ]

   [where search_conditions]

   [group by [all] aggregate_free_expression
       [, aggregate_free_expression]... ]
   [having search_conditions]

   [order by
   {[[[database.]owner.]{table_name.|view_name.}]
       column_name | select_list_number | expression}
           [asc | desc]
   [,{[[[database.]owner.]{table_name|view_name.}]
       column_name | select_list_number | expression}
           [asc | desc]]...]

   [compute row_aggregate(column_name)
           [, row_aggregate(column_name)]...
       [by column_name [, column_name]...]]

   [for {read only | update [of column_name_list]}]

   [for browse]
```

The keywords in the **select** statement, as in all other SQL statements, must be used in the order shown in the syntax statement.

Keywords and Options

all – includes all rows in the results. **all** is the default.

distinct – includes only unique rows in the results. Null values are considered equal for the purposes of the keyword **distinct**: only one NULL is selected no matter how many are encountered. **distinct** must be the first word in the select list.

select_list – is one or more of the following items:

- "*", representing all columns in **create table** order.
- A list of column names in the order in which you want to see them. When selecting an existing IDENTITY column, you can substitute the **syb_identity** keyword, qualified by the table name where necessary, for the actual column name.
- A specification to add a new IDENTITY column to the result table:

  ***column_name* = identity(*precision*)**

- A replacement for the default column heading (the column name), in the form:

  column_heading = column_name

  or:

  column_name column_heading

  or:

  column_name* as *column_heading

  The column heading may be enclosed in quotation marks for any of these forms. The heading must be enclosed in quotation marks if it is not a valid identifier (that is, if it is a reserved word, if it begins with a special character, or if it contains spaces or punctuation marks).

- An expression (a column name, constant, function, or any combination of column names, constants, and functions connected by arithmetic or bitwise operators, or a subquery).
- A built-in function or an aggregate
- Any combination of the items listed above

  The *select_list* can also perform variable assignment, in the form:

  @variable = expression
  [, *@variable = expression* ...]

  You cannot combine variable assignment with any of the other *select_list* options.

  For more information, see "Expressions", "Functions" and "Subqueries".

into – creates a new table based on the columns specified in the select list and the rows chosen in the **where** clause.

from – indicates which tables and views to use in the **select** statement. It is always required except when the select list contains no

column names (that is, constants and arithmetic expressions only):

```
select 5 x, 2 y, "the product is", 5*2 Result
```

At most, a query can reference 16 tables. This maximum includes:

- Tables (or views on tables) listed in the **from** clause
- Each instance of multiple references to the same table (self-joins)
- Tables referenced in subqueries
- Any work tables created as a result of the query (such as using vector aggregate functions with a **group by** clause)

table_name, view_name – lists tables and views used in the **select** statement. If there is more than one table or view in the list, separate their names by commas. The order of the tables and views after the keyword **from** does not affect the results.

You can query tables in different databases in the same statement.

Table names and view names can be given correlation names, either for clarity or to distinguish the different roles that a table or view play in a self-join or subquery. Give the table or view name, then a space, then the correlation name, like this:

```
select pub_name, title_id
    from publishers pu, titles t
    where t.pub_id = pu.pub_id
```

All other references to that table, for example in a **where** clause, must use the correlation name.

holdlock – makes a shared lock on a specified table or view more restrictive by holding it until the completion of a transaction (instead of releasing the shared lock as soon as the required data page is no longer needed, whether or not the transaction has been completed).

The **holdlock** option applies only to the table or view for which it is specified, and only for the duration of the transaction defined by the statement in which it is used. Setting **transaction isolation level 3** option of the **set** command implicitly applies a **holdlock** for each **select** within a transaction. The keyword **holdlock** is not permitted in a **select** statement that includes the **for browse** option. You cannot specify both a **holdlock** and **noholdlock** option in a query.

noholdlock – prevents the server from holding any locks acquired during the execution of this **select** statement, regardless of the transaction isolation level currently in effect. You cannot specify both a **holdlock** and **noholdlock** option in a query.

shared – instructs SQL Server to use a shared lock (instead of an update lock) on a specified table or view. This allows other clients to obtain an update lock on that table or view. You can use the **shared** keyword only with a **select** included as part of a **declare cursor** statement. For example:

```
declare shared_crsr cursor
for select title, title_id
from titles shared
where title_id like "BU%"
```

You can use the **holdlock** keyword in conjunction with **shared** after each table or view name, but **holdlock** must precede **shared**.

search_conditions – used to set the conditions for the rows that are retrieved. A search condition can include column names, expressions, arithmetic operators, comparison operators, the keywords **not**, **like**, **is null**, **and**, **or**, **between**, **in**, **exists**, **any**, and **all**, subqueries, or any combination of these items. Although there is no limit to the number of search conditions that can be included in a statement, a **where** clause can include a maximum of 128 search conditions per table. **where** search conditions and **having** search conditions are identical with one exception: **where** clauses cannot contain aggregates and **having** clauses can. See the sections on **where** Clause," "Expressions," and "Subqueries" for more information.

group by – finds a value for each group. These values appear as new columns in the results, rather than as new rows.

When **group by** is used with standard SQL, each item in the select list must either have a fixed value in every row in the group or be used with aggregate functions, which produce a single value for each group. Transact-SQL has no such restrictions on the items in the select list. Also, Transact-SQL allows you to group by any expression (though not by a column alias), while with standard SQL you can only group by a column.

You can use these aggregates with **group by** (*expression* is almost always a column name):

Aggregate Function	Result
sum([**all** | **distinct**] *expression*)	Total of the values in the numeric column
avg([**all** | **distinct**] *expression*)	Average of the values in the numeric column
count([**all** | **distinct**] *expression*)	Number of (distinct) non-null values in the column
count(*)	Number of selected rows
max(*expression*)	Highest value in the column
min(*expression*)	Lowest value in the column

Table 1-23: Results of using aggregates with group by

(See "**group by** and **having** Clauses" and "Aggregates" for more information.)

A table can be grouped by any combination of columns—that is, groups can be nested within each other. You cannot group by a column heading; you must use a column name, an expression, or a number representing the position of the item in the select list.

group by all – includes all groups in the results, even those that don't have any rows that meet the search conditions.

(See "**group by** and **having** Clauses" for an example.)

aggregate_free_expression – is an expression that includes no aggregates.

having – sets conditions for the **group by** clause, similar to the way that **where** sets conditions for the **select** clause. There is no limit on the number of conditions that can be included.

You can use a **having** clause without a **group by** clause.

If there are columns in the select list that neither have aggregate functions applied to them nor are included in the query's **group by** clause (illegal in standard SQL), the meanings of **having** and **where** are somewhat different.

In this situation, a **where** clause restricts the rows that are included in the calculation of the aggregate, but does not restrict the rows returned by the query. Conversely, a **having** clause

restricts the rows returned by the query, but does not affect the calculation of the aggregate. See "**group by** and **having** Clauses" for examples.

order by – sorts the results by columns. In Transact-SQL, you can **order by** items that do not appear in the select list. You can sort by a column name, a column heading (or alias), an expression, or a number representing the position of the item in the select list (the *select_list_number*). If you sort by select list number, the columns to which the **order by** clause refers must be included in the select list, and the select list cannot be * (asterisk).

asc – sorts results in ascending order (the default).

desc – sorts results in descending order.

compute – used with row aggregates (**sum**, **avg**, **min**, **max**, and **count**) to generate control break summary values. The summary values appear as additional rows in the query results, allowing you to see detail and summary rows with one statement.

You cannot use a **select into** clause with **compute**.

If you use **compute by**, you must also use an **order by** clause. The columns listed after **compute by** must be identical to or a subset of those listed after **order by**, and must be in the same left-to-right order, start with the same expression, and not skip any expressions.

For example, if the **order by** clause is:

```
order by a, b, c
```

the **compute by** clause can be any (or all) of these:

```
compute by a, b, c
compute by a, b
compute by a
```

The keyword **compute** can be used without **by** to generate grand totals, grand counts, etc. **order by** is optional if you use **compute** without **by**. (See "**compute** Clause" for details and examples.)

for {**read only** | **update**} – specifies that a cursor result set is read-only or updatable. You can use this option only within a stored procedure, and only when the procedure defines a query for a cursor. In this case, the **select** is the only statement allowed in the procedure, and it defines the **for read only** or **for update** option (instead of the **declare cursor** statement). This method of declaring

cursors provides the advantage of page-level locking while fetching rows.

If the **select** in the stored procedure is not used to define a cursor, SQL Server ignores this option. See the Embedded SQL documentation for more information about declaring cursors using stored procedures. For information about read-only or updatable cursors, see "Cursors" in Chapter 3, ''Transact-SQL Topics''.

of *column_name_list* – is the list of columns from a cursor result set defined as updatable with the **for update** option.

for browse – must be attached to the end of a SQL statement sent to SQL Server in a DB-Library browse application. (See "Browse Mode" and the *Open Client DB-Library Reference Manual* for details.)

Examples

```
1. select * from publishers

2. select pub_id, pub_name, city, state from
   publishers

3. select "The publisher's name is",
   Publisher = pub_name, pub_id
   from publishers

4. select type as Type, price as Price
   from titles

5. select pub_id, total = sum (total_sales)
       into #advance_rpt
   from titles
   where advance < $10000
       and total_sales is not null
   group by pub_id
   having count(*) > 1

6. select "Author_name" = au_fname + " " + au_lname
       into #tempnames
       from authors

7. select type, price, advance from titles
   order by type desc
   compute avg(price), sum(advance) by type
   compute sum(price), sum(advance)

8. select type, price, advance from titles
   compute sum(price), sum(advance)
```

```
9. select * into coffeetabletitles from titles
   where price > $20

10.select * into newtitles from titles
   where price > $25 and price < $20

11.select sales_east.syb_identity,
   sales_west.syb_identity
   from sales_east, sales_west

12.select *, row_id = identity(10)
   into newtitles from titles
```

Comments

- The keywords in the **select** statement, as in all other statements, must be used in the order shown in the syntax statement.
- The keyword **all** can be used after **select** for compatibility with other implementations of SQL. **all** is the default. Used in this context, **all** is the opposite of **distinct**: all retrieved rows are included in the results, whether or not some are duplicates.
- Except in **create table**, **create view**, and **select into** statements, column headings may include any characters, including blanks and SQL Server keywords, if the column heading is enclosed in quotes. If the heading is not enclosed in quotes, it must conform to the rules for identifiers.
- Column headings in **create table**, **create view**, and **select into** statements, table aliases, and **select** statements in stored procedures called by APT-SQL, must conform to the rules for identifiers.
- To insert data with **select** from a table that has null values in some fields into a table which does not allow null values, you must provide a substitute value for any NULL entries in the original table. For example, to insert data into an *advances* table that won't allow null values, this example substitutes "0" for the NULL fields:

  ```
  insert advances
  select pub_id, isnull(advance, 0) from titles
  ```

  Without the **isnull** function, this command would insert all the rows with non-null values into *advances*, and produce error messages for all the rows where the *advance* column in *titles* contained NULL.

If this kind of substitution cannot be made for your data, it is not possible to insert the data containing null values into the columns with the NOT NULL specification.

Two tables may be identically structured, but differ in whether null values are permitted in some fields. You can use **sp_help** to see the null types of the columns in your table.

- To select an IDENTITY column into a result table, include the column name (or the **syb_identity** keyword) in the **select** statement's *column_list*. The new column observes the following rules:
  - If an IDENTITY column is selected more than once, it is defined as NOT NULL in the new table. It does not inherit the IDENTITY property.
  - If an IDENTITY column is selected as part of an expression, the resulting column does not inherit the IDENTITY property. It is created as NULL if any column in the expression allows nulls, and NOT NULL otherwise.
  - If the select statement contains a **group by** clause or aggregate function, the resulting column does not inherit the IDENTITY property. Columns that include an aggregate of the IDENTITY column are created NULL; others, NOT NULL.
  - An IDENTITY column that is selected into a table with a union or join does not retain the IDENTITY property. If the table contains the union of the IDENTITY column and a NULL column, the new column is defined as NULL. Otherwise, it is defined as NOT NULL.
- The default length of *text* or *image* data returned with a **select** statement is 32K. Use **set textsize** to change the value. The size for the current session is stored in the global variable *@@textsize*. Certain client software may issue a **set textsize** command on logging into the server.
- Data from remote SQL Servers can be retrieved through the use of remote procedure calls. See **create procedure** and **execute** for more information.
- A **select** used in a cursor definition (through **declare cursor**) must contain a **from** clause, but it cannot contain the **compute**, **for browse**, or **into** clauses. If the **select** contains any of the following constructs:
  - **distinct** option
  - **group by** clause

- Aggregate functions
- **union** operator

the cursor is considered "read-only" and not "updatable". If you declare a cursor inside a stored procedure with a **select** containing an **order by** clause, that cursor is also considered read-only. Even if it is considered updatable, you cannot delete a row using a cursor which is defined by a **select** containing a join of two or more tables. See the section "**declare cursor**" for more information.

- The name of the new table must be unique in the database and must conform to the rules for identifiers. You can also select into temporary tables. See examples 5, 6 and 9.
- You can use **select into** to create a duplicate table with no data by having a false condition in the **where** clause. See example 10.
- You must provide a column heading for any column in the select list that contains an aggregate function or any expression. The use of any constant, arithmetic or character expression, the use of built-in functions, or concatenation in the select list requires a column heading for the affected item. The column heading must be a valid identifier, or must be enclosed in quotation marks. See examples 5 and 6.
- You cannot use **select into** inside a user-defined transaction, or in the same statement as a **compute** clause.
- **select into** is a two-step operation. The first step creates the new table. The second step inserts the specified rows into the new table. If the second step fails for any reason (hardware failure, exceeding a system resource, etc.), the entire statement is rolled back. All rows are deleted, and the new table is dropped.
- Any rules, constraints, or defaults associated with the base table are not carried over to the new table. You must bind those rules or defaults to the new table using **sp_bindrule** and **sp_bindefault**.
- The **select into/bulkcopy** option must be set on (by executing **sp_dboption**) in order to be able to **select into** a permanent table. You do not have to set the **select into/bulkcopy** option on in order to **select into** a temporary table, since the temporary database is never recovered.

  Once you have used **select into** in a database, you must perform a full database dump before you can use the **dump transaction** command. **select into** operations are not logged and changes are therefore not recoverable from transaction logs. In this situation,

issuing the **dump transaction** statement produces an error message instructing you to use **dump database** instead.

By default, the **select into/bulkcopy** option is off in newly created databases. To change the default situation, turn this option on in the *model* database.

- **select into** runs more slowly while a **dump database** is taking place.
- You cannot use **select into** to create a new table with multiple IDENTITY columns. If the **select** statement includes both an existing IDENTITY column and a new IDENTITY specification of the form *column_name* = identity(*precision*), the statement fails.
- For information about the Embedded SQL command **select into** ***host_var_list***, see the *Open Client Embedded SQL Reference Manual.*

Standards and Compliance

Standard	Compliance Level	Comments
SQL89	Compliant	The following are Transact-SQL extensions: • use of **select into** to create a new table • **compute** clauses • global and local variables • the **holdlock**, **noholdlock**, and **shared** keywords, • the use of "*column_heading* = *column_name*" • qualified table and column names • the use of **select** in a **for browse** clause • the use within the select list of columns that are not in the **group by** list and have no aggregate functions
SQL92	Entry level compliant	

Permissions

select permission defaults to the owner of the table or view, who can transfer it to other users.

See Also

Commands	**compute** Clause, **create trigger**, **delete**, **group by** and **having** Clauses, **insert**, **union** operator, **update**, **where** Clause
Functions	Aggregate Functions, Row Aggregate Functions
Topics	Expressions, Wildcard Characters
System procedures	**sp_dboption**

set

Function

Sets SQL Server query-processing options for the duration of the user's work session. Can be used to set some options inside a trigger or stored procedure.

Syntax

```
set ansinull {on | off}

set ansi_permissions {on | off}

set arithabort [arith_overflow | numeric_truncation]
   {on | off}

set arithignore [arith_overflow] {on | off}

set {chained, close on endtran, nocount, noexec,
   parseonly, procid, self_recursion, showplan}
   {on | off}

set char_convert {off | on [with {error | no_error}] |
   charset [with {error | no_error}]}

set cursor rows number for cursor_name

set {datefirst number, dateformat format,
   language language}

set dup_in_subquery {on | off}

set fipsflagger {on | off}

set flushmessage {on | off}

set identity_insert [database.[owner.]]table_name
   {on | off}

set offsets {select, from, order, compute, table,
   procedure, statement, param, execute} {on | off}

set quoted_identifier {on | off}

set role {"sa_role" | "sso_role" | "oper_role"}
   {on | off}

set {rowcount number, textsize number}

set statistics {io, time} {on | off}

set string_rtruncation {on | off}

set textsize {number}

set transaction isolation level {1 | 3}
```

Keywords and Options

ansinull – determines whether or not evaluation of NULL-valued operands in SQL equality (=) or inequality (!=) comparisons or aggregate functions, also called set functions, is ANSI-compliant. When set **on**, SQL Server generates a warning each time an aggregate function eliminates a null-valued operand from calculation. This option does not affect how SQL Server evaluates NULL values in other kinds of SQL statements, such as **create table**.

ansi_permissions - determines whether ANSI-compliant permissions requirements for **delete** and **update** statements are checked. The default value is off. The following table summarizes these requirements:

Command	Permissions Required with set ansi_permissions off	Permissions Required with set ansi_permissions on
update	• **update** permission on columns where values are being set	• **update** permission on columns where values are being set *And...* • **select** permission on all columns appearing in **where** clause • **select** permission on all columns on right side of **set** clause
delete	• **delete** permission on table	• **delete** permission on table from which rows are being deleted *And...* • **select** permission on all columns appearing in **where** clause

Table 1-24: Permissions required for update and delete

arithabort – determines how SQL Server behaves when an arithmetic error occurs. The two **arithabort** options, **arithabort arith_overflow** and **arithabort numeric_truncation**, handle different types of arithmetic errors. You can set each option independently, or set both options with a single **set arithabort on** or **set arithabort off** statement.

- **arithabort arith_overflow** specifies behavior following a divide-by-zero error or a loss of precision during either an explicit or an implicit datatype conversion. This type of error is considered serious. The default setting, **arithabort arith_overflow on**, rolls back the entire transaction or batch in which the error occurs. If you set **arithabort arith_overflow off**, SQL Server aborts the statement that causes the error but continues to process other statements in the transaction or batch.
- **arithabort numeric_truncation** specifies behavior following a loss of scale by an exact numeric type during an implicit datatype conversion. (When an explicit conversion results in a loss of scale, the results are truncated without warning.) The default setting, **arithabort numeric_truncation on**, aborts the statement that causes the error but continues to process other statements in the transaction or batch. If you set **arithabort numeric_truncation off**, SQL Server truncates the query results and continues processing.

arithignore arith_overflow– determines whether SQL Server displays a message after a divide-by-zero error or a loss of precision. The default setting, **off**, displays a warning message after these errors. Setting **arithignore arith_overflow on** suppresses warning messages after these errors. The optional **arith_overflow** keyword can be omitted without any effect.

chained – begins a transaction just before the first data retrieval or data modification statement at the beginning of a session and after a transaction ends. In chained mode, SQL Server implicitly executes a **begin transaction** before the following statements: **delete**, **fetch**, **insert**, **open**, **select**, and **update**. You cannot execute **set chained** within a transaction.

char_convert – turns character set conversion off and on between SQL Server and a client. If the client is using a 4.6 or later release of Open Client DB-Library, conversion is turned on during the login process (if the client and server use different character sets) and set to a default based on the character set that the client is using. **set char_convert** *charset* can also be used to start conversion between the server character set and a different client character set.

charset can be either the character set's ID or name from *syscharsets* where *type* is less than 2000.

set char_convert off turns conversion off so that characters are sent and received unchanged. **set char_convert on** turns conversion on if it is turned off. If character set conversion was not previously

turned on during the login process or by the **set char_convert** *charset* command, **set char_convert on** generates an error message.

When the **with no_error** option is included, SQL Server does not notify an application when characters from SQL Server cannot be converted to the client's character set. Error reporting is initially set on when a client connects with SQL Server: if you do not want error reporting, you must turn it off for each session with **set char_convert {on | charset} with no_error**. To turn error reporting back on within a session, use **set char_convert {on | charset} with error.**

Whether or not error reporting is turned on, the bytes which cannot be converted are replaced with ASCII question marks (?). (See the *System Administration Guide* for a more complete discussion of error handling in character set conversion.)

close on endtran– causes SQL Server to close all cursors opened within a transaction at the end of that transaction. A transaction ends by using either the Transact-SQL **commit** or **rollback** statements. However, only cursors declared within the scope (stored procedure, trigger, etc.) that sets this option are affected. See "Cursors" for more information cursor scopes.

cursor rows – causes SQL Server to return the *number* of rows for each cursor **fetch** request from a client application. You can **set** the **cursor rows** option for a cursor whether it is open or closed. However, this option does not affect a **fetch** request containing an **into** clause. *cursor_name* specifies the cursor for which to **set** the number of rows returned.

datefirst – sets the first week day to a number from 1 to 7. The us_english default is 1 (Sunday).

dateformat – sets the order of the date parts *month/day/year* for entering *datetime* or *smalldatetime* data. Valid arguments are *mdy*, *dmy*, *ymd*, *ydm*, *myd*, or *dym*. The us_english default is *mdy*.

dup_in_subquery – Controls whether a subquery using an **in** clause returns duplicate values. The default is off; duplicate rows are not returned. Prior to Release 10.0, SQL Server returned a row for each matching row in the subquery. ANSI specifies removing duplicate values from the result set. This option is provided as a workaround until you can re-write your queries for 10.0 behavior; it will not be supported in future releases.

fipsflagger – determines whether SQL Server displays a warning message when Transact-SQL extensions to entry level SQL92 are used. By default, SQL Server does not tell you when you use non-standard SQL.

flushmessage – determines when SQL Server returns messages to the user. By default messages are stored in a buffer until the query that generated them completes or the buffer is filled to capacity. Use **set flushmessage on** to return messages to the user immediately, as they are generated.

identity_insert – determines whether inserts into a table's IDENTITY column are allowed. (Note that updates to an IDENTITY column are never allowed.) This option can be used only with base tables. It cannot be used with views or set within a trigger.

Setting **identity_insert on** allows the table owner, Database Owner, or System Administrator to explicitly insert a value into an IDENTITY column. Setting **identity _insert off** restores the default behavior by prohibiting inserts to IDENTITY columns. At any given time, you can turn on the **identity_insert** option for a single table in a database.

Inserting a value into the IDENTITY column allows you to specify a "seed" value for the column or to restore a row that was deleted in error. Unless you have created a unique index on the IDENTITY column, SQL Server does not verify the uniqueness of the inserted value; you can insert any positive integer.

language – is the official name of the language that displays system messages. The language must be available on the server. us_english is the default.

nocount – turns off the display of rows affected by a statement. The global variable *@@rowcount* is updated even when **nocount** is on.

noexec – compiles each query but does not execute it. **noexec** is often used with **showplan**. Once **noexec** is turned on, no subsequent commands are executed (including other **set** commands) until **noexec** is turned off.

offsets – returns the position of specified keywords (with relation to the beginning of the query) in Transact-SQL statements. The keyword list is a comma-separated list that can include any of the following Transact-SQL constructs: **select**, **from**, **order**, **compute**, **table**, **procedure**, **statement**, **param**, and **execute**. This option is used in Open Client DB-Library only.

parseonly – checks the syntax of each query and returns any error messages without compiling or executing the query. Returns offsets if the **offsets** option is set on and there are no errors. **parseonly** should not be used inside a stored procedure or trigger.

procid – returns the ID number of the stored procedure to Open Client DB-Library / C (not to the user) before sending rows generated by the stored procedure.

quoted_identifier - determines whether SQL Server recognizes delimited identifiers. By default, **quoted_identifier** is set **off** and all identifiers must conform to the rules for valid identifiers. If you **set quoted_identifier on**, you can use table, view, and column names that begin with a non-alphabetic character, include characters that would not otherwise be allowed, or are reserved words by enclosing the identifiers within double quotation marks. Delimited identifiers cannot exceed 28 bytes, cannot be used as parameters to system procedures, and may not be recognized by all front-end products.

When **quoted_identifier** is **on**, all character strings enclosed within double quotes are treated as identifiers. Use single quotes around character or binary strings.

role – turns the specified role on or off during the current session. When you log in, all roles that have been granted to you are automatically turned on. Use **set role** to turn any roles off, and back on again if desired. The roles are **sa_role, sso_role,** and **oper_role**. If you are not a user in the current database, and if there is no "guest" user, you will be unable to turn off **sa_role**, because there is no server user ID for you to assume.

rowcount – causes SQL Server to stop processing the query (**select**, **insert**, **update** or **delete**) after the specified number of rows are affected. To turn this option off, use:

```
set rowcount 0
```

self_recursion – determines whether or not SQL Server allows triggers to cause themselves to fire again (called self recursion). By default, SQL Server does not allow self recursion in triggers. You can turn this option on only for the duration of a current client session; its effect is limited by the scope of the trigger that sets it. For example, if the trigger that sets **self_recursion on** returns or causes another trigger to fire, this option reverts to **off**.

showplan – generates a description of the processing plan for the query and immediately processes it unless **noexec** is set. This option may indicate whether or not an index is being used in a retrieval. The results of **showplan** are of use in performance diagnostics. Do not use **showplan** inside a stored procedure or trigger.

statistics io – displays the number of scans, the number of logical reads (pages accessed), and the number of physical reads (database device accesses) for each table referenced in the statement; for each command, it displays the number of pages written.

statistics time – displays the time it took to parse and compile for each command; for each step of the command, it displays the time it took to execute. Times are given in timeticks, the exact value of which is machine-dependent.

string_rtruncation– determines if SQL Server raises a SQLSTATE exception when an **insert** or **update** truncates a *char* or *varchar* string. If the truncated characters consist only of spaces, no exception is raised. The default setting, **off**, does not raise the SQLSTATE exception and the character string is silently truncated.

textsize – specifies the size in bytes of *text* or *image* type data to be returned with a **select** statement. The *@@textsize* global variable stores the current setting. To reset to the default size (32K), use the command:

```
set textsize 0
```

transaction isolation level – causes SQL Server to automatically apply a **holdlock** to all **select** operations in a transaction when set to isolation level 3. If you also set chained mode, that isolation level remains in effect for any data retrieval or modification statement that implicitly begins a transaction. By default, SQL Server's transaction isolation level is 1, which allows shared locks on data. You cannot change an isolation level within a transaction.

Examples

1.
```
set showplan, noexec on
go
select * from publishers
go
```

   For each query, SQL Server returns a description of the processing plan but does not execute it.

2. **set textsize 100**

   Sets the limit on *text* or *image* data returned with a **select** statement to 100 bytes.

3. **set rowcount 10**

   For each **insert**, **update**, **delete**, or **select**, SQL Server stops processing the query after it affects the first ten rows.

4. **set char_convert on with error**

   SQL Server turns character set conversion on, setting it to a default based on the character set the client is using. SQL Server also notifies the client or application when characters cannot be converted to the client's character set.

5. **set cursor rows 5 for test_cursor**

   Once set, SQL Server returns five rows for each succeeding **fetch** statement requested by a client using *test_cursor*.

6. ```
   set identity_insert stores_south on
   go
   insert stores_south (syb_identity)
   values (100)
   go
   set identity_insert stores_south off
   go
   ```

   Allows the table owner or the Database Owner to insert a value of 100 into the IDENTITY column of the *stores_south* table, then prohibits further inserts to this column. Note the use of the **syb_identity** keyword; SQL Server replaces this with the actual name of the IDENTITY column.

7. **set transaction isolation level 3**

   Automatically implements read-locks with each **select** in a transaction for the duration of that transaction.

8. **set role "sa_role" off**

   Deactivates the user's System Administrator role for the current session.

9. **set flipsflagger on**

   Tells SQL Server to display a warning message if you use a Transact-SQL extension. If you then use non-standard SQL, like this:

   ```
   use pubs2
   go
   ```
   ```

SQL Server displays:

```
SQL statement on line number 1 contains Non-ANSI
text. The error is caused due to the use of use
database.
```

10. set ansinull on

Tells SQL Server to evaluate NULL-valued operands of equality (=) and inequality (!=) comparisons and aggregate functions in compliance with the entry level SQL92 standard.

When **set ansinull** is **on**, aggregate functions and row aggregates raise the following SQLSTATE warning upon encountering null values in one or more columns or rows:

```
Warning - null value eliminated in set function
```

If the value of either of the equality or inequality operands is NULL, the comparison's result is UNKNOWN. For example, the following query returns no rows in **ansinull** mode:

```
select * from titles where price = null
```

If you set **ansinull** mode off:

```
set ansinull off
```

the same query returns rows in which *price* is NULL.

11. set string_rtruncation on

Causes SQL Server to generate an exception when truncating a *char* or *nchar* string. If an **insert** or **update** would truncate a string, SQL Server displays:

```
string data, right truncation
```

12.
```
set quoted_identifier on
go
create table "!*&strange_table"
("emp's_name"char(10), age int)
go
set quoted_identifier off
go
```

Tells SQL Server to treat any character string enclosed in double quotes as an identifier. The table name "!*&strange_table" and the column name "emp's_name" are legal identifier names while **quoted_identifier** is set **on.**

```
13.set dup_in_subquery on
   go
   select pub_name from publishers
       where pub_id in (select pub_id from titles)
   go
```

With **dup_in_subquery** set to **on**, this query returns 18 rows, one for each matching *pub_id* in *titles*. With the default setting, **off**, the query returns three rows, one for each *pub_id* in the outer query that matches a row in the inner query.

Comments

- If you use the **set** command inside a trigger or stored procedure, the option reverts to its former setting after the trigger or procedure executes.
- If you specify more than one option, the first syntax error causes all following options to be ignored. The options specified before the error are, however, executed, and the new option values are set.
- The **ansinull** option affects equality (=) and inequality (!=) comparisons, and aggregate functions (also called set functions). SQL92 requires that if either one of the two operands of an equality comparison is NULL, the result is UNKNOWN.

  Transact-SQL treats NULL values differently. If one of the operands is a column, parameter, or variable, and the other operand is the NULL constant or a parameter or variable whose value is NULL, the result is either TRUE or FALSE.
- The **set** options can be divided into these categories:
  - **parseonly**, **noexec**, **showplan**, **rowcount**, and **nocount** control the way a query is executed. It doesn't make sense to set both **parseonly** and **noexec** on. The default setting for **rowcount** is 0 (return all rows); the default for the others is **off**.
  - The **statistics** options display performance statistics after each query. The default setting for these options is **off**.
  - **arithabort** determines whether SQL Server aborts queries with arithmetic overflow and numeric truncation errors. **arithignore** determines whether SQL Server prints a warning message if a query results in an arithmetic overflow. By default, both options are turned **on**.

➤ ***Note***

The **arithabort** and **arithignore** options have been redefined for Release 10.0. If you use these options in your applications, examine them to be sure they are still producing the desired effect.

- **offsets** and **procid** are used in DB-Library to interpret results from SQL Server. The default setting for these options is **on.**
- **datefirst**, **dateformat**, and **language** affect date functions, date order, and message display. In the default language, us_english, **datefirst** is 7 (Saturday), **dateformat** is *mdy*, and messages are displayed in us_english. **set language** implies that SQL Server should use the first weekday and date format of the language it specifies, but it will not override an explicit **set datefirst** or **set dateformat** command issued earlier in the current session.
- **textsize** controls the size of *text* type data returned with a **select** statement. The default setting is 32K in **isql**. Some client software sets other default values. Setting **textsize** to 0 restores the default value of 32K.
- **char_convert** controls character set conversion between SQL Server and a client.
- **cursor rows** and **close on endtran** affect the way SQL Server handles cursors. The default setting for **cursor rows** with all cursors is 1. The default setting for **close on endtran** is **off**.
- **identity_insert** allows or prohibits inserts that affect a table's **IDENTITY** column.
- **chained** and **transaction isolation level** allow SQL Server to handle ANSI-compliant transactions.
- **self_recursion** allows SQL Server to handle triggers that cause themselves to fire. The default setting for **self_recursion** is **off**.
- **fipsflagger on** causes SQL Server to flag the use of nonstandard SQL.
- **string_rtruncation on** causes SQL Server to raise an exception error when truncating a *char* or *nchar* string.
- **quoted_identifier on** causes SQL Server to treat character strings enclosed in double quotes as identifiers.

- All **set** options except **showplan** and **char_convert** take effect immediately. **showplan** takes effect in the following batch. Here are two examples using **set showplan on**:

```
set showplan on
select * from publishers
go
pub_id  pub_name                 city        state
------- ---------------------- ----------- -----
0736    New Age Books            Boston      MA
0877    Binnet & Hardley         Washington  DC
1389    Algodata Infosystems     Berkeley    CA

(3 rows affected)
```

But:

```
set showplan on
go
select * from publishers
go
STEP 1
The type of query is select
from table
publishers
Nested iteration
Table Scan

pub_id  pub_name                 city        state
------  --------------------     ----------- ----
0736    New Age Books            Boston      MA
0877    Binnet & Hardley         Washington  DC
1389    Algodata Infosystems     Berkeley    CA

(3 rows affected)
```

- When you log in, all roles granted to you are automatically enabled. Use **set role** to turn any of these options off or back on again. For example, if you have been granted the System Administrator role, you assume the identity (and user ID) of Database Owner within the current database. If you wish to assume your "real" user identity, execute this command:

```
set role "sa_role" off
```

If you are not a user in the current database, and if there is no "guest" user, you will be unable to turn off **sa_role**.

SQL92 Compliance

The SQL92 standard specifies behavior that differs from Transact-SQL behavior in earlier SQL Server releases. Compliant behavior is enabled by default for all Embedded-SQL precompiler applications. Other applications needing to match this standard behavior can use the following **set** options.

Option	Setting
ansi_permissions	**on**
ansinull	**on**
arithabort	**off**
arithabort numeric_truncation	**on**
arithignore	**off**
chained	**on**
close on endtran	**on**
fipsflagger	**on**
quoted_identifier	**on**
string_rtruncation	**on**
transaction isolation level	**3**

Table 1-25: set options for entry level SQL92 compliance

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

See Also

Commands	**create trigger**, **fetch**, **insert**
Functions	Datatype Conversion Functions
Topics	Batch Queries, Cursors, Datatypes, Roles, Transactions

setuser

Function

Allows a Database Owner to impersonate another user.

Syntax

```
setuser ["user_name"]
```

Examples

```
setuser "mary"
go
grant select on authors to joe
setuser
go
```

The Database Owner temporarily adopts Mary's identity in the database in order to grant Joe permissions on *authors*, a table owned by Mary.

Comments

- The Database Owner uses **setuser** to adopt the identity of another user in order to use another user's database object, to grant permissions, to create an object, or for some other reason.
- When the Database Owner uses the **setuser** command, SQL Server checks the permissions of the user being impersonated instead of the permissions of the Database Owner. The user being impersonated must be listed in the *sysusers* table of the database.
- The **setuser** command remains in effect until another **setuser** command is given, or until the current database is changed with the **use** command.
- Executing the **setuser** command with no user name re-establishes the Database Owner's original identity.
- System Administrators can use **setuser** to create objects that will be owned by another user. However, since a System Administrator operates outside the permissions system, she or he cannot use **setuser** to acquire another user's permissions.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

Permissions

setuser permission defaults to the Database Owner, and is not transferable.

See Also

Commands	**grant, revoke, use**

shutdown

Function

Shuts down the SQL Server from which the command is issued, its local Backup Server, or a remote Backup Server. This command can only be issued by a System Administrator.

Syntax

```
shutdown [srvname] [with {wait | nowait}]
```

Keywords and Options

srvname – is the logical name by which the Backup Server is known in the SQL Server's *sysservers* system table. This parameter is not required when shutting down the local SQL Server.

with wait– is the default. This brings the SQL Server or Backup Server down gracefully.

with nowait– shuts down the SQL Server or Backup Server immediately, without waiting for currently executing statements to finish.

Examples

1. **shutdown**

   Gracefully shuts down the SQL Server from which the **shutdown** command is issued.

2. **shutdown with nowait**

   Shuts down the SQL Server immediately.

3. **shutdown SYB_BACKUP**

   Shuts down the local Backup Server.

4. sp_helpserver

```
name        network_name   status                                id
----------  -------------  ----------------------------------- --
REM_BACKUP  whale_backup   timeouts, no net password encryption 3
SYB_BACKUP  slug_backup    timeouts, net password encryption    1
eel         eel                                                0
whale       whale         timeouts, no net password encryption 2

(return status = 0)
```

shutdown REM_BACKUP

Uses **sp_helpserver** to determine the *whale_backup* Backup Server's *name* within SQL Server. Shuts down this remote Backup Server by specifying this name in the **shutdown** command.

Comments

- Unless you use the **nowait** option, **shutdown** attempts to bring SQL Server down gracefully by:
  - Disabling logins (except for the System Administrator)
  - Performing a checkpoint in every database
  - Waiting for currently executing SQL statements or stored procedures to finish.

  Shutting down the server gracefully (without the **nowait** option) minimizes the amount of work that must be done by the automatic recovery process.
- Unless you use the **nowait** option, **shutdown** *backup_server* waits for active dumps and/or loads to complete. Once you issue a **shutdown** command to a Backup Server, no new dumps or loads that use this Backup Server can start.
- Use **shutdown with nowait** only in extreme circumstances. On SQL Server, issue a **checkpoint** command before executing a **shutdown with nowait.**
- You can only halt the local SQL Server with **shutdown**; you cannot halt a remote SQL Server.
- You can only halt a Backup Server if:
  - It is listed in your *sysservers* table. The system procedure **sp_addserver** adds entries to *sysservers.*
  - It is listed in the interfaces file for the SQL Server where you execute the command.

- Use the **sp_helpserver** system procedure to determine the name by which a Backup Server is known to the SQL Server. Specify the Backup Server's *name*—not its *network_name*—as **shutdown**'s *srv_name* parameter.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

Permissions

shutdown permission defaults to System Administrators, and is not transferable.

See Also

Commands	**checkpoint**
System procedures	**sp_addserver, sp_helpserver**

truncate table

Function

Removes all rows from a table.

Syntax

```
truncate table [[database.]owner.]table_name
```

Examples

```
truncate table authors
```

Removes all data in the *authors* table.

Comments

- **truncate table** deletes all rows from a table. The table structure and all of the indexes continue to exist until you issue a **drop table** command. The rules and defaults that are bound to the columns remain bound, and triggers remain in effect.
- **truncate table** deallocates the distribution pages for all indexes; remember to run **update statistics** after adding new rows to the table.
- **truncate table** is equivalent to—but faster than—a **delete** without a **where** clause. **delete** removes rows one at a time and logs each deleted row as a transaction; **truncate table** deallocates whole data pages and makes fewer log entries. Both **delete** and **truncate table** reclaim the space occupied by the data and its associated indexes.
- Although a **truncate table** statement is, in effect, like a **delete** without a **where** clause, **truncate table** cannot "fire" a trigger, because the deletion of the individual rows is not logged.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

Permissions

truncate table permission defaults to the table owner, and is not transferable.

See Also

Commands	**create trigger, delete, drop table**

union Operator

Function

Returns a single result set that combines the results of two or more queries. Duplicate rows are eliminated from the result set unless the **all** keyword is specified.

Syntax

```
    select select_list [into clause]
        [from clause] [where clause]
        [group by clause] [having clause]
[union [all]
    select select_list
        [from clause] [where clause]
        [group by clause] [having clause] ]...
[order by clause]
[compute clause]
```

Keywords and Options

union – creates the union of data specified by two **select** statements.

all – includes all rows in the results; duplicates are not removed.

into – creates a new table based on the columns specified in the select list and the rows chosen in the **where** clause. The first query in the union operation is the only one that can contain an **into** clause.

Examples

1.
```
select stor_id, stor_name from sales
union
select stor_id, stor_name from sales_east
```

   The result set includes the contents of the *stor_id* and *stor_name* columns of both the *sales* and *sales_east* tables.

2.
```
select pub_id, pub_name, city into results
from publishers
union
select stor_id, stor_name, city from stores
union
select stor_id, stor_name, city from stores_east
```

   The **into** clause in the first query specifies that table *results* hold the final result set of the union of the specified columns of the *publishers*, *stores*, and *stores_east* tables.

3.
```
select au_lname, city, state from authors
union
((select stor_name, city, state from sales
union
select stor_name, city, state from sales_east)
union
select pub_name, city, state from publishers)
```

   First the **union** of the specified columns in the *sales* and *sales_east* tables is generated, and then the **union** of that result with *publishers* is generated. Finally, the **union** of the second result and *authors* is generated.

Comments

- **order by** and **compute** clauses are allowed only at the end of the **union** statement to define the order of the final results or to compute summary values.
- **group by** and **having** clauses can be used only within individual queries and cannot be used to affect the final result set.
- The default evaluation order of a SQL statement containing **union** operators is left to right.
- Since **union** is a binary operation, parentheses must be added to an expression involving more than two queries to specify valuation order.
- The first query in the **union** statement may contain an **into** clause that creates a table to hold the final result set. The **into** statement must be in the first query or you will receive an error message. (See Example 2.)
- The **union** operator cannot appear within a **create view** statement.
- The **union** operator cannot appear within a subquery.
- The **for browse** clause cannot be specified in queries involving the **union** operator.
- The **union** operator can appear within an **insert-select** statement. For example:

  ```
  insert into sales.all
    select * from sales
    union
    select * from sales_east
  ```

- All select lists in the SQL statement must have the same number of expressions (column names, arithmetic expressions, aggregate

functions, etc.). For example, this statement is invalid because the first select list is longer than the second:

```
select au_id, title_id, au_ord from titleauthor
union
select stor_id, date from sales
```

- Corresponding columns in the select lists of **union** statements must occur in the same order, because **union** compares the columns one-to-one in the order given in the individual queries.
- The column names in the table resulting from a **union** are taken from the **first** individual query in the **union** statement. If you want to define a new column heading for the result set, you must do it in the first query. In addition, if you want to refer to a column in the result set by a new name, for example in an **order by** statement, it must be referred to in that way in the first **select** statement. For example, the following query is correct:

```
select Cities = city from stores
union
select city from stores_east
order by Cities
```

- You cannot use **union** in the **select** statement of an updatable cursor.
- The descriptions of the columns that are part of a **union** operation do not have to be identical. Here are the rules for comparing the datatypes and options of corresponding columns:

Datatype of Column in union Operation	Datatype of Corresponding Column in Result Table
Not datatype-compatible (data conversion not handled implicitly by SQL Server).	Error returned by SQL Server.
Both fixed length *char* with lengths L1 and L2.	Fixed length *char* with length equal to the greater of L1 and L2.
Both fixed length *binary* with lengths L1 and L2	Fixed length *binary* with length equal to the greater of L1 and L2.
Either or both are variable length *char*.	Variable length *char* with length equal to the maximum of the lengths specified for the column in the union.
Either or both variable length *binary*.	Variable length *binary* with length equal to the maximum of the lengths specified for the columns in the union.

Table 1-26: Comparing datatypes in union operations

Datatype of Column in union Operation	Datatype of Corresponding Column in Result Table
Both are numeric datatypes (for example, *smallint, int, float, money*).	A datatype equal to the maximum precision of the two columns. For example, if a column of table *A* is of type *int* and the corresponding column of table B *is* of type *float*, then the datatype of the corresponding column of the result table is *float*, because *float* is more precise than *int*.
Both columns' descriptions specify NOT NULL.	Specifies NOT NULL.

Table 1-26: Comparing datatypes in union operations (continued)

Standards and Compliance

Standard	Compliance Level	Comments
SQL89	Compliant	The following are Transact-SQL extensions: • The use of **union** in the **select** clause of an **insert** statement • Specifying new column headings in the **order by** clause of a **select** statement when the **union** operator is present in the **select** statement
SQL92	Entry level compliant	

See Also

Commands	**compute Clause, declare, group by and having Clauses, order by Clause, select, where Clause**
Functions	Datatype Conversion Functions
Topics	Datatypes

update

Function

Changes data in existing rows, either by adding data or by modifying existing data.

Syntax

```
update [[database.]owner.]{table_name | view_name}
   set [[[database.]owner.]{table_name.|view_name.}]
       column_name1 =
           {expression1|NULL|(select_statement)}
       [, column_name2 =
           {expression2|NULL|(select_statement)}]...
   [from [[database.]owner.]{table_name | view_name}
       [,[[database.]owner.]{table_name|view_name}]...]
   [where search_conditions]

update [[database.]owner.]{table_name | view_name}
   set [[[database.]owner.]{table_name.|view_name.}]
       column_name1 =
           {expression1|NULL|(select_statement)}
       [, column_name2 =
           {expression2|NULL|(select_statement)}]...
   where current of cursor_name
```

Keywords and Options

set – specifies the column name and assigns the new value. The value can be an expression or a NULL. When more than one column name and value pair is listed, they must be separated by commas.

from – uses data from other tables or views to modify rows in the table or view you are updating.

where – is a standard **where** clause. (See "**where** Clause.")

where current of – causes SQL Server to update the row of the table or view indicated by the current cursor position for *cursor_name*.

Examples

1.
```
update authors
set au_lname = "MacBadden"
where au_lname = "McBadden"
```

   All the McBaddens in the *authors* table are now MacBaddens.

```
2. update titles
   set total_sales = total_sales + qty
   from titles, salesdetail, sales
   where titles.title_id = salesdetail.title_id
       and salesdetail.stor_id = sales.stor_id
       and salesdetail.ord_num = sales.ord_num
       and sales.date in
           (select max(sales.date) from sales)
```

Modifies the *total_sales* column to reflect the most recent sales recorded in the *sales* and *salesdetail* tables. This assumes that only one set of sales is recorded for a given title on a given date, and that updates are up to date!

```
3. update titles
   set price = 24.95
   where current of title_crsr
```

Changes the price of the book in *titles* currently pointed at by *title_crsr* to $24.95.

```
4. update titles
   set price = 18.95
   where syb_identity = 4
```

Finds the row for which the IDENTITY column equals 4 and changes the price of the book to $18.95. SQL Server replaces the **syb_identity** keyword with the actual name of the IDENTITY column.

Comments

- Use **update** to change values in rows that have already been inserted. Use **insert** to add new rows.
- **update** interacts with the **ignore_dup_key, ignore_dup_row,** and **allow_dup_row** options set with the **create index** command. (See "**create index**" for more information.)
- SQL Server treats two different designations for the same table in an **update** as two tables. For example, the following **update** issued in *pubs2* specifies *discounts* as two tables (*discounts* and *pubs2..discounts)*:

```
update discounts
set discount = 25
from pubs2..discounts, pubs2..stores
where pubs2..discounts.stor_id =
    pubs2..stores.stor_id
```

In this case, the join does not include *discounts*, so the **where** condition remains true for every row; SQL Server updates all rows in *discounts* (which is not the desired result). To avoid this problem, use the same designation for a table throughout the statement.

- You can define a trigger that takes a specified action when an **update** command is issued on a specified table or on a specified column of a table.

update *and Transactions*

- When you set chained transaction mode on, and there is no transaction currently active, SQL Server implicitly begins a transaction with the **update** statement. To complete the update, you must either **commit** the transaction or **rollback** the changes. For example:

```
update stores set city = 'Concord'
    where stor_id = '7066'
if exists (select t1.city, t2.city
    from stores t1, stores t2
    where t1.city = t2.city
    and t1.state = t2.state
    and t1.stor_id < t2.stor_id)
        rollback transaction
else
        commit transaction
```

  This batch begins a transaction (using chained transaction mode) and updates a row in the *stores* table. If it updates a row containing the same city and state information as another store in the table, it rolls back the changes to *stores* and ends the transaction. Otherwise, it commits the updates and ends the transaction. For more information about the chained mode, see "Transactions".

- SQL Server does not prevent you from issuing an **update** statement that updates a single row more than once in a given transaction. However, because of the way the **update** is processed, updates from a single statement do not accumulate. That is, if an **update** statement modifies the same row twice, the second update is not based on the new values from the first update but on the original values. The results are unpredictable since they depend on the order of processing.

Updating Character Data

- Updating variable-length character data or *text* columns with the empty string ("") inserts a single space. Fixed-length character columns are padded to the defined length.
- All trailing spaces are removed from variable-length column data, except in the case of a string which contains only spaces. Strings that contain only spaces are truncated to a single space. Strings longer than the specified length of a *char, nchar, varchar,* or *nvarchar* column are silently truncated unless the **string_rtruncation** option is set to **on**.
- An **update** to a *text* column initializes the *text* column, assigns it a valid text pointer, and allocates at least one 2K data page.

update *and Cursors*

- To update a row using a cursor, first define the cursor with **declare cursor**, then **open** it. The cursor name cannot be a Transact-SQL parameter or local variable, and it must be an updatable cursor or SQL Server returns an error. Any update to the cursor result set also affects the base table row that the cursor row is derived from.
- The *table_name* or *view_name* specified with an **update...where current of** must be the table or view specified in the first **from** clause of the **select** statement that defines the cursor. If that **from** clause references more than one table or view (using a join), you can specify only the table or view actually being updated.

  After the update, the cursor position remains unchanged. You can continue to update the row at that cursor position as long as another SQL statement does not move the position of that cursor.
- SQL Server allows you to update columns that are not specified in the list of columns of the cursor's *select_statement*, but that are part of the tables specified in the *select_statement*. However, when you specify a *column_name_list* with **for update** when declaring the cursor, you can only update those specific columns.

Updating IDENTITY Columns

- A column with the IDENTITY property cannot be updated, either through its base table or through a view. To determine whether a column was defined with the IDENTITY property, use the **sp_help** system procedure on the column's base table.

- An IDENTITY column selected into a result table observes the following rules with regard to inheritance of the IDENTITY property:
  - If an IDENTITY column is selected more than once, it is defined as NOT NULL in the new table. It does not inherit the IDENTITY property.
  - If an IDENTITY column is selected as part of an expression, the resulting column does not inherit the IDENTITY property. It is created as NULL if any column in the expression allows nulls, and NOT NULL otherwise.
  - If the select statement contains a **group by** clause or aggregate function, the resulting column does not inherit the IDENTITY property. Columns that include an aggregate of the IDENTITY column are created NULL; others, NOT NULL.
  - An IDENTITY column that is selected into a table with a union or join does not retain the IDENTITY property. If the table contains the union of the IDENTITY column and a NULL column, the new column is defined as NULL. Otherwise, it is defined as NOT NULL.

Updating Data Through Views

- You cannot **update** views defined with the **distinct** clause.
- If a view is created **with check option**, each row that is updated through the view must remain visible through the view. For example, the *stores_cal* view includes all rows of the *stores* table for which *state* has a value of "CA". The **with check option** clause checks each **update** statement against the view's selection criteria:

```
create view stores_cal
as select * from stores
where state = "CA"
with check option
```

  update statements, such as the one below, fail if they change *state* to any value other than "CA":

```
update stores_cal
set state = "WA"
where store_id = "7066"
```

- If a view is created **with check option**, all views derived from the "base" view must satisfy the view's selection criteria. Each row updated through a derived view must remain visible through the base view.

Consider the view *stores_cal30*, which is derived from *stores_cal*. The new view includes information about stores in California with payment terms of "Net 30":

```
create view stores_cal30
as select * from stores_cal
where payterms = "Net 30"
```

Because *stores_cal* was created **with check option**, all rows updated through *stores_cal30* must remain visible through *stores_cal*. Any row that changes *state* to a value other than "CA" is rejected.

Notice that *stores_cal30* does not have a **with check option** clause of its own. This means that it is possible to update a row with a *payterms* value other than "Net 30" through *stores_cal30*. The following **update** statement would be successful, even though the row would no longer be visible through *stores_cal30*:

```
update stores_cal30
set payterms = "Net 60"
where stor_id = "7067"
```

- You cannot update a row through a view that joins columns from two or more tables unless both of the following conditions are true:
  - The view has no **with check option** clause
  - All columns being updated belong to the same base table
- **update** statements are allowed on join views **with check option**. The update fails if any of the affected columns appears in the **where** clause, in an expression that includes columns from more than one table.
- If you update a row through a join view, all affected columns must belong to the same base table.

Standards and Compliance

Standard	Compliance Level	Comments
SQL89	Compliant	The use of a **from** clause or a qualified table or column name are Transact-SQL extensions detected by the FIPS flagger. Updates through a join view or a view whose target list contains an expression are Transact-SQL extensions that cannot be detected until runtime and are not flagged by the FIPS flagger.
SQL92	Entry level compliant	

Permissions

update permission defaults to the table or view owner, who can transfer it to other users.

If you have **set ansi_permissions on**, you will need, in addition to the regular permissions required for update statements, to have **select** permission on all columns appearing in the **where** clause, and **select** permission on all columns on the right side of the **set** clause.

See Also

Commands	**create default**, **create index**, **create rule**, **create trigger**, **insert**, **where** Clause
Topics	Cursors, Text and Image Datatypes, Transactions,
System procedures	**sp_bindefault**, **sp_bindrule**, **sp_help**, **sp_unbindefault**, **sp_unbindrule**

update statistics

Function

Updates information about the distribution of key values in specified indexes.

Syntax

```
update statistics table_name [index_name]
```

Keywords and Options

table_name – is the name of the table with which the index is associated. *table_name* is required since Transact-SQL does not require index names to be unique in a database.

index_name – is the name of the index to be updated. If an index name is not specified, the distribution statistics for all the indexes in the specified table are updated.

Comments

- SQL Server keeps statistics about the distribution of the key values in each index, and uses these statistics in its decisions about which index(es) to use in query processing. **update statistics** is automatically run when you create or recreate an index on a table that already contains data.
- The optimization of your queries depends on the accuracy of the distribution steps. If there is significant change in the key values in your index, you should re-run **update statistics** on that index. Use the **update statistics** command if a great deal of data in an indexed column has been added, changed, or removed (that is, if you suspect that the distribution of key values has changed).
- **update statistics** updates allocation page values used to estimate the number of rows in a table. These statistics are used by the **rowcnt** function and **sp_spaceused**.
- Run **update statistics** after adding new rows to a table whose rows had previously been deleted with **truncate table**.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

Permissions

update statistics permission defaults to the table owner and is not transferable. The command can also be executed by the Database Owner, who can impersonate the table owner by running the **setuser** command.

See Also

Commands	**create index**
System procedures	**sp_helpindex**

use

Function

Specifies the database with which you want to work.

Syntax

use ***database_name***

Examples

```
use pubs2
go
```

The current database is now *pubs2*.

Comments

- The **use** command must be executed before you can reference objects in a database.
- **use** cannot be included in a stored procedure or a trigger.
- An alias permits a user to use a database under another name in order to gain access to that database. Use the system procedure **sp_addalias**.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

Permissions

Permission for the **use** command defaults to all users. Only valid users of the named database are granted access to it.

See Also

Commands	create database, drop database
System procedures	sp_addalias, sp_adduser, sp_modifylogin

waitfor

Function

Specifies a specific time, a time interval, or an event for the execution of a statement block, stored procedure, or transaction.

Syntax

```
waitfor { delay time | time time | errorexit
   | processexit | mirrorexit }
```

Keywords and Options

delay – instructs SQL Server to wait until the specified amount of time has passed, up to a maximum of 24 hours.

time– instructs SQL Server to wait until the specified time.

time – a time in one of the acceptable formats for *datetime* data. You cannot specify dates—the date portion of the *datetime* value is not allowed.

errorexit – instructs SQL Server to wait until a kernel or user process terminates abnormally.

processexit – instructs SQL Server to wait until a kernel or user process terminates for any reason.

mirrorexit – instructs SQL Server to wait for a mirror failure.

Examples

```
1. begin
       waitfor time "14:20"
       insert chess(next_move)
           values('Q-KR5')
       execute sendmail 'judy'
   end
```

At 2:20 p.m., the *chess* table will be updated with my next move, and a procedure called *sendmail* will insert a row in a table owned by Judy, notifying her that a new move now exists in the *chess* table.

```
2. begin
       waitfor delay "00:00:10"
       print "Ten seconds have passed.  Your time
           is up."
   end
```

Comments

- After issuing the **waitfor** command, you cannot use your connection to SQL Server until the time or event that you specified occurs.
- You can use **waitfor errorexit** with a procedure that kills the abnormally terminated process, in order to free system resources that would otherwise be taken up by an infected process.
- To find out which process terminated, check the *sysprocesses* table with the system procedure **sp_who**.
- The time you specify with **waitfor time** or **waitfor delay** can include hours, minutes, and seconds. Use the format *hh:mi:ss*, as described in "Datatypes."

  For example:

  ```
  waitfor time "16:23"
  ```

  instructs SQL Server to wait for 4:23 p.m. The statement:

  ```
  waitfor delay "01:30"
  ```

  instructs SQL Server to wait one hour and 30 minutes.
- You can use **waitfor mirrorexit** within a DB-Library program to notify users when there is a mirror failure.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

See Also

Commands	**begin...end**
Topics	Datatypes, Disk Mirroring
System procedures	**sp_who**

where Clause

Function

Sets the search conditions in a **select**, **insert**, **update**, or **delete** statement. (Joins and subqueries are specified in the search conditions: see the "Joins" and "Subqueries" sections for full details.)

Syntax

Search conditions immediately follow the keyword **where** in a **select**, **insert**, **update**, or **delete** statement. If you use more than one of the search conditions in a single statement, connect the conditions with **and** or **or**.

```
where [not] expression comparison_operator expression

where [not] expression [not] like  "match_string"
   [escape "escape_character"]

where [not] expression is [not] null

where [not]
   expression [not] between expression and expression

where [not]
   expression [not] in ({value_list | subquery})

where [not] exists (subquery)

where [not]
   expression comparison_operator {any|all} (subquery)

where [not] column_name join_operator column_name

where [not] boolean_expression

where [not] expression {and | or} [not] expression
```

Keywords and Options

all – is used with **>** or **<** and a subquery. It returns results when all values retrieved in the subquery match the value in the **where** or **having** clause of the outer statement. (See "Subqueries" for more information.)

and – joins two conditions and returns results when both of the conditions are true.

When more than one logical operator is used in a statement, **and** operators are normally evaluated first. However, you can change the order of execution with parentheses.

any – is used with **>**, **<**, or **=** and a subquery. It returns results when any value retrieved in the subquery matches the value in the **where** or **having** clause of the outer statement. (See "Subqueries" for more information.)

between – is the range-start keyword. Use **and** for the range-end value. The range:

```
where @val between x and y
```

is inclusive; the range:

```
where @val > x and @val < y
```

is not. Queries using **between** return no rows if the first value specified is greater than the second value.

column_name – is the name of the column used in the comparison. Qualify the column name with its table or view name if there is any ambiguity. For columns with the IDENTITY property, you can specify the **syb_identity** keyword, qualified by a table name where necessary, rather than the actual column name.

comparison_operator – is one of the following:

Symbol	Meaning
=	equal to
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
!=	not equal to
<>	not equal to
!>	not greater than
!<	not less than

Table 1-27: Comparison operators

In comparing *char*, *nchar*, *varchar*, and *nvarchar* data, **<** means closer to the beginning of the alphabet and **>** means closer to the end of the alphabet.

Case and special character evaluations depend on the collating sequence of the operating system on the machine on which SQL Server is located. For example, lowercase letters may be greater than uppercase letters, and uppercase letters may be greater than numbers.

Trailing blanks are ignored for the purposes of comparison. For example, "Dirk" is the same as "Dirk ".

In comparing dates, **<** means earlier and **>** means later. Put quotes around all character and date data used with a comparison operator. For example:

```
= "Bennet"
> "94609"
```

See "Datatypes" for more information about data entry rules.

escape – specifies an escape character with which you can search for literal occurrences of wildcard characters. See "Wildcard Characters" for a complete explanation.

exists – is used with a subquery to test for the existence of some result from the subquery. (See "Subqueries" for more information.)

expression – is a column name, a constant, a function, a subquery, or any combination of column names, constants, and functions connected by arithmetic or bitwise operators.

The arithmetic operators are:

Symbol	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo

Table 1-28: Arithmetic operators

Addition, subtraction, division, and multiplication can be used on exact numeric, approximate numeric, and money types. The modulo operator can only be used with integer types. A modulo is the integer remainder after a division involving two integers. For example, 21 % 9 = 3 because 21 divided by 9 equals 2 with a remainder of 3.

The bitwise operators are:

Symbol	Meaning
&	Bitwise and (two operands)
|	Bitwise or (two operands)
^	Bitwise exclusive or (two operands)
~	Bitwise not (one operand)

Table 1-29: Bitwise operators

The bitwise operators can only be used on the integer types. All the bitwise operators translate the integer arguments into binary representation before evaluating them. (See "Expressions" for more information.)

in – allows you to select values that match any one of a list of values. The comparator can be a constant or a column name, and the list can be a set of constants or, more commonly, a subquery. (See "Subqueries" for information on using **in** with a subquery.) Enclose the list of values in parentheses.

is null – is used when searching for null values.

join_operator – is a comparison operator or one of the symbols =* or *=. (See "Joins" for more information.)

like – is a keyword indicating that the following character string (enclosed by single or double quotes) is a matching pattern. **like** is available for *char*, *varchar*, *nchar*, *nvarchar*, and *datetime* columns (but not to search for seconds or milliseconds).

You can use the keyword **like** and wildcard characters with *datetime* data as well as with *char* and *varchar*. When you use **like** with *datetime* values, SQL Server converts the dates to the standard *datetime* format, and then to *varchar*. Since the standard storage format doesn't include seconds or milliseconds, you cannot search for seconds or milliseconds with **like** and a pattern.

It is a good idea to use **like** when you search for *datetime* values, since *datetime* entries may contain a variety of date parts. For example, if you insert the value "9:20" into a column named *arrival_time*, the clause:

```
where arrival_time = '9:20'
```

would not find it because SQL Server converts the entry into "Jan 1, 1900 9:20AM." However, the clause:

```
where arrival_time like '%9:20%'
```

would find it.

logical_expression – is an expression that returns "true" or "false". (See "Expressions" for a full definition.)

match_string – is a string of characters and wildcards enclosed in quotes. The wildcards are:

Symbol	Meaning
%	any string of 0 or more characters
_	any single character
[ ]	any single character within the specified range ([a-f]) or set ([abcdef])
[^]	any single character not within the specified range ([^a-f]) or set ([^abcdef])

Table 1-30: Wildcard operators

For complete information, including information on using alternative character set definitions in wildcards, see "Wildcard Characters."

not – negates any logical expression or keywords such as **like**, **null**, **between**, **in**, and **exists**.

or – joins two conditions and returns results when either of the conditions is true.

When more than one logical operator is used in a statement, **or** operators are normally evaluated after **and** operators. However, you can change the order of execution with parentheses.

subquery – is a restricted **select** statement (**order by** and **compute** clauses and the keyword **into** are not allowed) inside the **where** or **having** clause of a **select**, **insert**, **delete**, **update**, or subquery. (See "Subqueries" for more information.)

value_list – is a list of values. Put single or double quotes around character values, and separate each value from the following one with a comma. (See example 7.) The list can be a list of variables, for example:

```
in (@a, @b, @c)
```

but you cannot use a variable containing a list, such as:

```
@a = "'1', '2', '3'"
```

for a values list.

Examples

1. `where advance * $2 > total_sales * price`
2. `where phone not like '415%'`

   Finds all the rows in which the phone number does not begin with 415.

3. `where au_lname like "[CK]ars[eo]n"`

   Finds the rows for authors named Carson, Carsen, Karsen, and Karson.

4. `where sales_east.syb_identity = 4`

   Finds the row of the *sales_east* table in which the IDENTITY column has a value of 4.

5. `where advance < $5000 or advance is null`
6. `where (type = "business" or type = "psychology") and advance > $5500`
7. `where total_sales between 4095 and 12000`
8. `where state in ('CA', 'IN', ''MD')`

   Finds the rows in which the state is one of the three in the list.

Comments

- **where** and **having** search conditions are identical, except that aggregate functions are not permitted in **where** clauses. For example:

  ```
  having avg(price) > 20
  ```

  is legal;

  ```
  where avg(price) > 20
  ```

  is not. See "Aggregates" for information on the use of aggregate functions, and "**group by** and **having** Clauses" for examples.

- There are two ways to specify literal quotes within a *char* or *varchar* entry. The first method is to use two quotes. For example, if you began a character entry with a single quote and wish to include a single quote as part of the entry, use two single quotes:

  ```
  'I don''t understand.''
  ```

  With double quotes:

  ```
  "He said, ""It's not really confusing."""
  ```

  The second method is to enclose a quote in the opposite kind of quote mark. In other words, surround an entry containing

double quotes with single quotes (or vice versa). Here are some examples:

```
'George said, "There must be a better way."'
"Isn't there a better way?"
'George asked, "Isn"t there a better way?"'
```

- To enter a character string longer than the width of your screen, enter a backslash (\) before going to the next line.
- If a column is compared to a constant or variable in a **where** clause, SQL Server converts the constant or variable into the datatype of the column, so the system can use the index for data retrieval. For example, *float* expressions are converted to *int* when compared to an *int* column. For example:

  ```
  where int_column = 2
  ```

  selects rows where *int_column* = 2.
- A **where** clause can include a maximum of 128 search arguments per table, where the search arguments are in the form:

  column_name comparison-operator constant-expression

Standards and Compliance

Standard	Compliance Level
SQL89	Compliant
SQL92	Entry level compliant

See Also

Commands	**delete**, **execute**, **group by** and **having** Clauses, **insert**, **select**, **update**
Topics	Datatypes, Expressions, Subqueries, Wildcard Characters
System procedures	**sp_helpjoins**

while

Function

Sets a condition for the repeated execution of a statement or statement block. The statement(s) are executed repeatedly as long as the specified condition is true.

Syntax

```
while logical_expression
      statement
```

Keywords and Options

logical_expression – is any expression that returns TRUE, FALSE or NULL.

statement – can be a single SQL statement, but is usually a block of SQL statements delimited by **begin** and **end.**

Examples

```
while (select avg(price) from titles) < $30
begin
    update titles
        set price = price * 2
    select avg(price) from titles
end
```

If the average price is less than $30, double the prices of all books in the *titles* table. Then, you can query for the new average price. As long as it is still less than $30, the **while** loop keeps doubling the prices. Other than determining the average price after each update, the **select** inside the **while** loop also indicates how many loops where completed (each average result returned by SQL Server indicates one loop).

Comments

- The execution of statements in the **while** loop can be controlled from inside the loop with the **break** and **continue** commands.
- **continue** causes the **while** loop to restart, skipping any statements after the **continue. break** causes an exit from the **while** loop, and any statements that appear after the keyword **end**, which marks the end of the loop, are executed. **break** and **continue** are often (but not always) activated by **if** tests.

For example:

```
while (select avg(price) from titles) < $30
begin
    update titles
        set price = price * 2
    if (select max(price) from titles) > $50
        break
    else
        if (select avg(price) from titles) > $30
            continue
    print "Average price still under $30"
end

select title_id, price from titles
    where price > $30
```

This batch continues to double the prices of all books in *titles* as long as the average book price is less than $30. However, if any book price exceeds $50, the **break** stops the **while** loop. The **continue** prevents the **print** statement from executing if the average does exceed $30. Regardless of how the **while** loop terminates (either normally or through the **break**), the last query indicates what books are priced over $30.

- If two or more **while** loops are nested, the **break** exits to the next outermost loop. First all the statements after the end of the inner loop run, and then the next outermost loop restarts.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

See Also

Commands	**begin...end, break, continue, goto label**

writetext

Function

Permits non-logged, interactive updating of an existing *text* or *image* column.

Syntax

```
writetext [[database.]owner.]table_name.column_name
   text_pointer [with log] data
```

Keywords and Options

table_name.column_name – the name of the column must include the table name. The database name and owner name are optional.

text_pointer– a *varbinary(16)* value that stores the pointer to the *text* or *image* data. Use the **textptr** function to determine this value, as shown in the example below. *text* and *image* data is not stored in the same set of linked pages as other table columns. It is stored in a separate set of linked pages. A pointer to the actual location is stored with the data; **textptr** returns this pointer.

with log – logs the inserted *text* or *image* data. The use of this option aids media recovery, but logging large blocks of data quickly increases the size of the transaction log. Make sure that the transaction log resides on a separate database device (see **create database**, **sp_logdevice**, and the *System Administration Guide* for details).

data – the data to be written into the *text* or *image* column. *text* data must be enclosed in quotes. *image* data must be preceded by "0x". Check the information about the client software you are using to determine the maximum length of *text* or *image* data the client can handle.

Examples

```
declare @val varbinary(16)
select @val = textptr(copy) from blurbs
    where au_id = "409-56-7008"
writetext blurbs.copy @val with log "hello world"
```

This example puts the text pointer into the local variable *@val*, then **writetext** places the text string "hello world" into the text field pointed to by *@val*.

Comments

- The maximum length of text that can be inserted interactively with **writetext** is approximately 120K bytes for *text* and *image* data.
- By default, **writetext** is a non-logged operation. This means that *text* or *image* data is not logged when it is written into the database. In order to use **writetext** in its default, non-logged state, a System Administrator must use **sp_dboption** to set **select into/bulkcopy** on. This permits the insertion of non-logged data.
- **writetext** updates *text* data in an existing row. The update completely replaces all of the existing text.
- **writetext** operations are not caught by an **insert** or **update** trigger.
- **writetext** requires a valid text pointer to the *text* or *image* column. In order for a valid text pointer to exist, a *text* column must contain either actual data or a null value that has been explicitly entered with **update**.

  Given the table *textnull* with columns *key* and *x*, where *x* is a *text* column that permits nulls, this **update** sets all the *text* values to NULL and assigns a valid text pointer in the *text* column:

  ```
  update textnull
  set x = null
  ```

  No text pointer results from an **insert** of an explicit null:

  ```
  insert textnull values (2,null)
  ```

  or this implicit null insert:

  ```
  insert textnull (key)
  values (2)
  ```

- **insert** and **update** on *text* columns are logged operations.
- You cannot use **writetext** on *text* and *image* columns in views.
- If you attempt to use **writetext** on *text* values after changing to a multibyte character set and have not run **dbcc fix_text**, the command fails and an error message is generated instructing you to run **dbcc fix_text** on the table.
- **writetext** in its default. unlogged mode runs more slowly while a **dump database** is taking place.
- The Client Library functions **dbwritetext** and **dbmoretext** are faster and use less dynamic memory than **writetext**. These functions can insert up to two gigabytes of *text* data.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

See Also

Commands	**readtext**
Functions	text and image Functions
Topics	text and image Datatypes

Functions

2 Transact-SQL Functions

This chapter describes the Transact-SQL functions. Functions are used to return information from the database. They are allowed in the **select list**, in the **where** clause, and anywhere else an expression is allowed. They are often used as part of a stored procedure or program.

Table 2-1 lists the different types of Transact-SQL functions and describes the type of information each returns:

Type of Function	Description
Aggregate	Generate summary values that appear as new columns in the query results.
Datatype Conversion	Change expressions from one datatype to another and specify new display formats for date/time information.
Date	Do computations on *datetime* and *smalldatetime* values and their components, date parts.
Mathematical	Return values commonly needed for operations on mathematical data.
Row Aggregate	Generate summary values that appear as additional rows in the query results.
String	Operate on binary data, character strings, and expressions.
System	Return special information from the database.
text/*image*	Supply values commonly needed for operations on *text* and *image* data.

Table 2-1: Transact-SQL functions

Aggregate Functions

Function

The aggregate functions generate summary values that appear as new columns in the query results. They can be used in the select list or the **having** clause of a **select** statement or subquery, and often appear in a statement that includes a **group by** clause.

Syntax

Aggregates follow the general form:

aggregate_function **([all | distinct]** ***expression*****)**

The aggregate functions, their individual syntax, and the results they produce are shown later. Note that *expression* is usually a column name, **all** and **distinct** are optional for **sum**, **avg**, and **count** (but not **count(*)**), and that **count(*)** takes no arguments. **sum** and **avg** can only be used on exact numeric, approximate numeric, and money datatypes.

Aggregate Function	Result
sum([all | distinct] *expression*)	Total of (distinct) values in the numeric column
avg([all | distinct] *expression*)	Average of (distinct) values in the numeric column
count([all | distinct] *expression*)	Number of (distinct) non-null values in the column
count(*)	Number of selected rows
max(*expression*)	Highest value in the expression
min(*expression*)	Lowest value in the expression

Table 2-2: Aggregate functions

Keywords and Options

aggregate_function – is the name of one of the aggregates. The aggregate functions calculate summary values, such as averages and sums, from the values in a particular column. For each set of rows to which an aggregate function is applied, a single value is generated.

The aggregates can be used in a select list or a **having** clause. They cannot be used in a **where** clause.

Aggregates are often used with **group by**. With **group by**, the table is divided into groups. Aggregates produce a single value for each group. Without **group by**, an aggregate function in the select list produces a single value as a result, whether it is operating on all the rows in a table or on a subset of rows defined by a **where** clause.

all – applies the aggregate to all values. **all** is the default.

distinct – eliminates duplicate values before an aggregate is applied. **distinct** is optional with **sum**, **avg**, and **count**.

column_name – is the name of a column.

expression – is a column name, constant, function, any combination of column names, constants, and functions connected by arithmetic or bitwise operators, or a subquery. With aggregates, an expression is usually a column name. (See "Expressions" for more information.)

sum – finds the sum of all the values in a column. **sum** can only be used on numeric (integer, floating point, or money) datatypes. Null values are ignored in calculating sums.

avg – finds the average of the values in a column. **sum** can only be used on numeric (integer, floating point, or money) datatypes. Null values are ignored in calculating averages.

count – finds the number of non-null values in a column. When **distinct** is specified, **count** finds the number of unique non-null values. **count** can be used with all datatypes except *text* and *image*. Null values are ignored when counting. **count(***column_name)* will return a value of 0 on empty tables, on columns that contain only null values, and on groups that contain only null values.

count(*) – finds the number of rows. **count(*)** does not take any arguments, and cannot be used with **distinct**. All rows are counted, regardless of the presence of null values.

max – finds the maximum value in a column. **max** can be used with exact and approximate numeric, character, and *datetime* columns. It cannot be used with *bit* columns. With character columns, **max** finds the highest value in the collating sequence. **max** ignores null values. **distinct** is not available, since it is not meaningful with **max**.

min – finds the minimum value in a column. **min** can be used with numeric, character, and *datetime* columns. It cannot be used with

bit columns. With character columns, **min** finds the lowest value in the sort sequence. **min** ignores null values. **distinct** is not available, since it is not meaningful with **min**.

Examples

1.
```
select avg(advance), sum(total_sales)
from titles
where type = "business"
```

   Calculates the average advance and the sum of total sales for all business books. Each of these aggregate functions produces a single summary value for all of the retrieved rows.

2.
```
select type, avg(advance), sum(total_sales)
from titles
group by type
```

   Used with a **group by** clause, the aggregate functions produce single values for each group, rather than for the whole table. This statement produces summary values for each type of book.

3.
```
select count(distinct city)
from authors
```

   Finds the number of different cities in which authors live.

4.
```
select type
from titles
group by type
having count(*) > 1
```

   Lists the types in the *titles* table, but eliminates the types that include only one book, or none.

5.
```
select pub_id, sum(advance), avg(price)
from titles
group by pub_id
having sum(advance) > $25000 and avg(price) > $15
```

   Groups the *titles* table by publishers, and includes only those groups of publishers who have paid more than $25,000 in total advances, and whose books average more than $15 in price.

Comments

- Aggregate functions can be used in the *select_list* or in the **having** clause of a **select** statement. They cannot be used in a **where** clause. Because each aggregate in a query requires its own work table, a query using aggregates can exceed the maximum number of tables allowed in a query (16).

- Aggregate functions calculate the summary values of the non-null values in a particular column. If the **ansinull** option is set **off** (the default), there is no warning when an aggregate function encounters a null. If **ansinull** is set **on**, a query returns the following SQLSTATE warning when an aggregate function encounters a null:

  ```
  Warning- null value eliminated in set function
  ```

- Aggregate functions can be applied to all the rows in a table, in which case they produce a single value, a scalar aggregate. They can also be applied to all the rows that have the same value in a specified column or expression (using the **group by** and, optionally, the **having** clause), in which case they produce a value for each group, a vector aggregate. The results of the aggregate functions are shown as new columns.

  You can nest a vector aggregate inside a scalar aggregate. For example:

  ```
  select type, avg(price), avg(avg(price))
  from titles
  group by type
  ```

  ```
  type
  ------------  ------------  ------------
  UNDECIDED             NULL        15.23
  business             13.73        15.23
  mod_cook             11.49        15.23
  popular_comp         21.48        15.23
  psychology           13.50        15.23
  trad_cook            15.96        15.23

  (6 rows affected)
  ```

  The **group by** clause applies to the vector aggregate—in this case, **avg**(*price*). The scalar aggregate—**avg**(**avg***(price*))—is the average of the average prices by type in the *titles* table.

- In standard SQL, when a *select_list* includes an aggregate, all the *select_list* columns must either have aggregate functions applied to them or be in the **group by** list. Transact-SQL has no such restrictions.

  The first example following shows a **select** statement with the standard restrictions; the second one shows the same statement with another item (*title_id*) added to the select list; **order by** is also added to illustrate the difference in displays. These "extra" columns can also be referenced in a **having** clause.

```
select type, avg(price), avg(advance)
from titles
group by type
```

```
type
------------  ------------  ------------
UNDECIDED             NULL          NULL
business             13.73     6,281.25
mod_cook             11.49     7,500.00
popular_comp         21.48     7,500.00
psychology           13.50     4,255.00
trad_cook            15.96     6,333.33

(6 rows affected)
```

```
select type, title_id, avg(price), avg(advance)
from titles
group by type
order by type
```

```
type        title_id
----------- --------    ----------- ---------
UNDECIDED       MC3026      NULL          NULL
business        BU1032      13.73    6,281.25
business        BU1111      13.73    6,281.25
business        BU2075      13.73    6,281.25
business        BU7832      13.73    6,281.25
mod_cook        MC2222      11.49    7,500.00
mod_cook        MC3021      11.49    7,500.00
popular_comp    PC1035      21.48    7,500.00
popular_comp    PC8888      21.48    7,500.00
popular_comp    PC9999      21.48    7,500.00
psychology      PS1372      13.50    4,255.00
psychology      PS2091      13.50    4,255.00
psychology      PS2106      13.50    4,255.00
psychology      PS3333      13.50    4,255.00
psychology      PS7777      13.50    4,255.00
trad_cook       TC3218      15.96    6,333.33
trad_cook       TC4203      15.96    6,333.33
trad_cook       TC7777      15.96    6,333.33

(18 rows affected)
```

- You can use either a column name or any other expression (except a column heading or alias) after **group by.**
- Null values in the **group by** column are put into a single group.
- The **compute** clause in a **select** statement uses row aggregates to produce summary values. The row aggregates make it possible to

retrieve detail and summary rows with one command. The following example illustrates this feature:

```
select type, title_id, price, advance
from titles
where type = "psychology"
order by type
compute sum(price), sum(advance) by type
```

```
type        title_id    price       advance
----------- --------    ----------  ---------
psychology  PS1372          21.59   7,000.00
psychology  PS2091          10.95   2,275.00
psychology  PS2106          7.00    6,000.00
psychology  PS3333          19.99   2,000.00
psychology  PS7777          7.99    4,000.00
                            sum     sum
                            -------  ----------
                            67.52   21,275.00

(6 rows affected)
```

Note the difference in display between the above example and the earlier examples without **compute**.

- When tables are being joined, including **count(*)** in the **select** list produces the count of the number of rows in the joined results. If the objective is to count the number of rows from one table that match criteria, use **count**(*column_name*).
- When you sum or average integer data, SQL Server treats the result as an *int* value, even if the datatype of the column is *smallint* or *tinyint*. To avoid overflow errors in DB-Library programs, declare all variables for results of averages or sums as type *int*.
- Aggregate functions cannot be used on virtual tables such as *sysprocesses* and *syslocks*.
- If you include an aggregate function in the **select** clause of a cursor, that cursor is not updatable.

Standards and Compliance

Standard	Compliance Level	Comments
SQL89	Compliant	By default, the aggregate functions are not compliant. **set ansinull on** for compliant behavior.
SQL92	Entry level compliant	

See Also

Commands	**compute** Clause, **group by** and **having** Clauses, **select**, **where** Clause
Functions	Row Aggregate Functions
Topics	Cursors

Datatype Conversion Functions

Function

Datatype conversion functions change expressions from one datatype to another and specify new display formats for date / time information. SQL Server provides three datatype conversion functions, **convert()**, **inttohex()**, and **hextoint()**, which can be used in the select list, in the **where** clause, and anywhere else an expression is allowed.

Syntax

Function	Argument	Result
convert	(*datatype* [(*length*) | (*precision*[, *scale*])], *expression*[, *style*])	Converts between a wide variety of datatypes and reformats date / time and money data for display purposes.
hextoint	(*hexadecimal_string*)	Returns the platform-independent integer equivalent of a hexadecimal string.
inttohex	(*integer_expression*)	Returns the platform-independent hexadecimal equivalent of an integer.

Table 2-3: Datatype conversion functions

Keywords and Options

datatype – is the system-supplied datatype (for example, *char(10)*, *varbinary(50)*, or *int*) into which the expression is to be converted. User-defined datatypes cannot be used.

length – is an optional parameter used with *char*, *nchar*, *varchar*, *nvarchar*, *binary* and *varbinary* datatypes. If you do not supply a length, SQL Server truncates the data to 30 characters for the character types and 30 bytes for the binary types. The maximum allowable length for character and binary data is 255.

precision – is the number of significant digits in a *numeric* or *decimal* datatype. For *float* datatypes, precision is the number of significant binary digits in the mantissa. If you do not supply a precision, SQL Server uses the default precision of 18 for *numeric* and *decimal* datatypes.

scale – is the number of digits to the right of the decimal point in a *numeric*, or *decimal* datatype. If you do not supply a scale, SQL Server uses the default scale of 0.

expression – is the value to be converted from one datatype or date format to another.

hexadecimal_string – is the hexadecimal value to be converted to an integer. This must be either a a character type column or variable name or a valid hexadecimal string, with or without a 0x prefix, enclosed in quotes.

integer_expression – is the integer value to be converted to a hexadecimal string.

style – is the display format to use for the converted data. When converting *money* or *smallmoney* data to a character type, use a *style* of 1 to display a comma after every three digits.

When converting *datetime* or *smalldatetime* data to a character type, use the style numbers in Table 2-4 to specify the display format. Values in the leftmost column display two-digit years (yy). For four-digit years (yyyy), add 100 or use the value in the middle column.

Without Century (yy)	With Century (yyyy)	Output
-	0 or 100	mon dd yyyy hh:miAM (or PM)
1	101	mm/dd/yy
2	102	yy.mm.dd
3	103	dd/mm/yy
4	104	dd.mm.yy
5	105	dd-mm-yy
6	106	dd mon yy
7	107	mon dd, yy
8	108	hh:mm:ss
-	9 or 109	mon dd yyyy hh:mi:ss:mmmAM (or PM)
10	110	mm-dd-yy

Table 2-4: Display formats for date/time information

Without Century (yy)	With Century (yyyy)	Output
11	111	yy/mm/dd
12	112	yymmdd

Table 2-4: Display formats for date/time information (continued)

The default values (*style* 0 or 100, and 9 or 109) always return the century (yyyy).

When converting to *char* or *varchar* from *smalldatetime*, styles that include seconds or milliseconds show zeroes in those positions.

Supported Conversions

SQL Server performs certain datatype conversions automatically. These are called **implicit conversions**. For example, if you compare a *char* expression and a *datetime* expression, or a *smallint* expression and an *int* expression, or *char* expressions of different lengths, SQL Server automatically converts one datatype to another.

You must request other datatype conversions explicitly, using one of the built-in datatype conversion functions. For example, before concatenating numeric expressions, you must convert them to character expressions.

SQL Server does not allow you to convert certain datatypes to certain other datatypes, either implicitly or explicitly. For example, you cannot convert *smallint* data to *datetime*, or *datetime* data to *smallint*. Unsupported conversions result in error messages.

Table 2-5 indicates whether individual datatype conversions are performed implicitly or explicitly or are unsupported.

From / To	*tinyint*	*smallint*	*int*	*decimal*	*numeric*	*real*	*float*	*char, nchar*	*varchar, nvarchar*	*text*	*smallmoney*	*money*	*bit*	*smalldatetime*	*datetime*	*binary*	*varbinary*	*image*
tinyint	-	I	I	I	I	I	I	E	E	U	I	I	I	U	U	I	I	U
smallint	I	-	I	I	I	I	I	E	E	U	I	I	I	U	U	I	I	U
int	I	I	-	I	I	I	I	E	E	U	I	I	I	U	U	I	I	U
decimal	I	I	I	I/E	I/E	I	I	E	E	U	I	I	I	U	U	I	I	U
numeric	I	I	I	I/E	I/E	I	I	E	E	U	I	I	I	U	U	I	I	U
real	I	I	I	I	I	-	I	E	E	U	I	I	I	U	U	I	I	U
float	I	I	I	I	I	I	-	E	E	U	I	I	I	U	U	I	I	U
char, nchar	E	E	E	E	E	E	E	I	I	I	E	E	E	I	I	E	E	I
varchar, nvarchar	E	E	E	E	E	E	E	I	I	I	E	E	E	I	I	E	E	I
text	U	U	U	U	U	U	U	E	E	U	U	U	U	U	U	U	U	U
smallmoney	I	I	I	I	I	I	I	I	I	U	-	I	I	U	U	I	I	U
money	I	I	I	I	I	I	I	I	I	U	I	-	I	U	U	I	I	U
bit	I	I	I	I	I	I	I	I	I	U	I	I	-	U	U	I	I	U
smalldatetime	U	U	U	U	U	U	U	E	E	U	U	U	U	-	I	I	I	U
datetime	U	U	U	U	U	U	U	E	E	U	U	U	U	I	-	I	I	U
binary	I	I	I	I	I	I	I	I	I	U	I	I	I	I	I	-	I	I
varbinary	I	I	I	I	I	I	I	I	I	U	I	I	I	I	I	I	-	I
image	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	I	I	U

Key:

E Explicit datatype conversion is required.

I Conversion can be done either implicitly or with an explicit datatype conversion function.

I/E Explicit datatype conversion function required when there is loss of precision or scale and arithabort numeric_truncation is on; implicit conversion allowed otherwise.

U Unsupported conversion.

- Conversion of a datatype to itself. In general, these conversions are allowed but meaningless.

Table 2-5: Explicit, implicit, and unsupported datatype conversions

Examples

1. **`select title, convert(char(12), total_sales) from titles`**
2. **`select title, total_sales from titles where convert(char(20), total_sales) like "1%"`**
3. **`select convert(char(12), getdate(), 3)`**

   Converts the current date to style "3", *dd/mm/yy*.
4. **`select convert(varchar(12), pubdate, 3) from titles`**

   If the value *pubdate* can be null, you must use *varchar* rather than *char*, or errors may result.
5. **`select hextoint ("0x00000100")`**

   Returns the integer equivalent of the hexadecimal string 0x00000100. The result is always 256, regardless of the platform on which it is executed.
6. **`select convert(integer, 0x00000100)`**

   Returns the integer equivalent of the string 0x00000100. Results can vary from one platform to another.
7. **`select inttohex (10)`**

   Returns a hexadecimal string, without 0x prefix, representing the value of the integer 10. The result is always "0000000A" regardless of the executing platform.
8. **`select convert (binary, 10)`**

   Returns the platform-specific bit pattern as a Sybase binary type.
9. **`select convert(bit, $1.11)`**

   Returns 1, the bit string equivalent of $1.11.

Converting Character Data to a Non-Character Type

Character data can be converted to a non-character type—such as a money, date/time, exact numeric, or approximate numeric type—if it consists entirely of characters that are valid for the new type. Leading blanks are ignored.

Syntax errors are generated when the data includes unacceptable characters. Following are some examples of characters that cause syntax errors:

- Commas or decimal points in integer data
- Commas in monetary data

- Letters in exact or approximate numeric data or bit stream data
- Misspelled month names in date/time data

Converting from One Character Type to Another

When converting from a multibyte character set to a single-byte character set, characters with no single-byte equivalent are converted to blanks.

text columns can be explicitly converted to *char, nchar, varchar,* or *nvarchar.* You are limited to the maximum length of the character datatypes, 255 bytes. If you do not specify the length, the converted value has a default length of 30 bytes.

Converting Numbers to a Character Type

Exact and approximate numeric data can be converted to a character type. If the new type is too short to accommodate the entire string, an insufficient space error is generated. For example, the following conversion tries to store a 5-character string in a 1-character type:

```
select convert(char(1), 12.34)
Insufficient result space for explicit conversion
of NUMERIC value '12.34' to a CHAR field.
```

➤ ***Note***

When converting *float* data to a character type, the new type should be at least 25 characters long.

Rounding During Conversion to/from Money Types

The *money* and *smallmoney* types store four digits to the right of the decimal point, but round up to the nearest hundredth (.01) for display purposes. When data is converted to a money type, it is rounded up to four places.

Data converted from a money type follows the same rounding behavior if possible. If the new type is an exact numeric with less than three decimal places, the data is rounded to the scale of the new type. For example, when $4.50 is converted to an integer, it yields 4:

```
select convert(int, $4.50)
 -----------
           5
```

Data converted to *money* or *smallmoney* is assumed to be in full currency units, such as dollars, rather than in fractional units, such as

cents. For example, the integer value of 5 would be converted to the money equivalent of 5 dollars, not 5 cents, in us_english.

Converting Date/time Information

Data that is recognizable as a date can be converted to *datetime* or *smalldatetime*. Incorrect month names lead to syntax errors. Dates that fall outside the acceptable range for the datatype lead to arithmetic overflow errors.

When *datetime* values are converted to *smalldatetime*, they are rounded up to the nearest minute.

Converting Between Numeric Types

Data can be converted from one numeric type to another. If the new type is an exact numeric whose precision or scale is not sufficient to hold the data, errors can occur. Use the **arithabort** and **arithignore** options to determine how these errors are handled.

➤ ***Note***

The **arithabort** and **arithignore** options have been redefined for Release 10.0. If you use these options in your applications, examine them to be sure they are still producing the desired behavior.

Arithmetic Overflow and Divide-by Zero Errors

Divide-by-zero errors occur when SQL Server tries to divide a numeric value by zero. Arithmetic overflow errors occur when the new type has too few decimal places to accommodate the results. This happens during

- Explicit or implicit conversions to exact types with a lower precision or scale
- Explicit or implicit conversions of data that falls outside the acceptable range for a money or datetime type
- Conversions of hexadecimal strings requiring more than 4 bytes of storage using **hextoint**

Both arithmetic overflow and divide-by-zero errors are considered serious, whether they occur during implicit or explicit conversions. Use the **arithabort arith_overflow** option to determine how SQL Server handles these errors. The default setting, **arithabort arith_overflow on**, rolls back the entire transaction or batch in which the error occurs. If you set **arithabort arith_overflow off**, SQL Server aborts the statement that

causes the error but continues to process other statements in the transaction or batch.You can use the *@@error* global variable to check statement results.

Use the **arithignore arith_overflow** option to determine whether SQL Server displays a message after these errors. The default setting, **off**, displays a warning message when a divide-by-zero error or a loss of precision occurs. Setting **arithignore arith_overflow on** suppresses warning messages after these errors. The optional **arith_overflow** keyword can be omitted without any effect.

Scale Errors

When an explicit conversion results in a loss of scale, the results are truncated without warning. For example, when you explicitly convert a *float*, *numeric*, or *decimal* type to an *integer*, SQL Server assumes you really want the result to be an integer and truncates all numbers to the right of the decimal point.

During implicit conversions to *numeric* or *decimal* types, loss of scale generates a scale error. Use the **arithabort numeric_truncation** option to determine how serious such an error is considered. The default setting, **arithabort numeric_truncation on**, aborts the statement that causes the error but continues to process other statements in the transaction or batch. If you set **arithabort numeric_truncation off**, SQL Server truncates the query results and continues processing.

➤ ***Note***

For entry level SQL92 compliance, set:
- arithabort arith_overflow off
- arithabort numeric_truncation on
- arithignore off

Domain Errors

The **convert()** function generates a domain error when the function's argument falls outside the range over which the function is defined. This should happen very rarely.

Conversions between Binary and Integer Types

The *binary* and *varbinary* types store hexadecimal-like data consisting of a 0x prefix followed by a string of digits and letters. These strings are interpreted differently by different platforms. For example, the string 0x0000100 represents 65536 on machines that consider byte 0

most significant, and 256 on machines that consider byte 0 least significant.

The convert Function and Implicit Conversions

The binary types can be converted to integer types either explicitly, using the **convert** function, or implicitly. The data is stripped of its 0x prefix and then zero-padded if it is too short for the new type or truncated if it is too long.

Both **convert** and the implicit datatype conversions evaluate binary data differently on different platforms. Because of this, results may vary from one platform to another. Use the **hextoint** function for platform-independent conversion of hexadecimal strings to integers, and the **inttohex** function for platform-independent conversion of integers to hexadecimal values.

The hextoint Function

Use the **hextoint** function for platform-independent conversions of hexadecimal data to integers. **hextoint** accepts a valid hexadecimal string, with or without a 0x prefix, enclosed in quotes, or the name of a character type column or variable.

hextoint returns the integer equivalent of the hexadecimal string. The function always returns the same integer equivalent for a given hexadecimal string, regardless of the platform on which it is executed.

The inttohex Function

Use the **inttohex** function for platform-independent conversions of integers to hexadecimal strings. **inttohex** accepts any expression that evaluates to an integer. It always returns the same hexadecimal equivalent for a given expression, regardless of the platform on which it is executed.

Converting Image Columns to Binary Types

You can use the **convert** function to convert an *image* column to *binary* or *varbinary*. You are limited to the maximum length of the *binary* datatypes, 255 bytes. If you do not specify the length, the converted value has a default length of 30 characters.

Converting Other Types to bit

Exact and approximate numeric types can be converted to the bit type implicitly. Character types require an explicit **convert** function.

The expression being converted must consist only of digits, a decimal point, a currency symbol, and a plus or minus sign. The presence of other characters generates syntax errors.

The *bit* equivalent of 0 is 0. The *bit* equivalent of any other number is 1.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

See Also

Functions	Date Functions, Mathematical Functions, String Functions, text and image Functions
Topics	Datatypes

Date Functions

Function

Manipulate *datetime* values.

Syntax

All date functions except **getdate** take arguments. Function names, arguments, and results are listed in the following table.

Function	Argument	Result
getdate	()	Returns the current system date and time.
datename	(*datepart*, *date*)	Returns the name of the specified part (such as the month "June") of a *datetime* value, as a character string. If the result is numeric, such as "23" for the day, it is still returned as a character string.
datepart	(*datepart*, *date*)	Returns an integer value for the specified part of a *datetime* value.
datediff	(*datepart*, *date1*, *date2*)	Returns *date2* - *date1*, measured in the specified date part.
dateadd	(*datepart*, *numeric_expression*, *date*)	Returns the date produced by adding the specified number of the specified date parts to the date. *numeric_expression* can be any numeric type; the value is truncated to an integer.

Table 2-6: Date functions

Keywords and Options

getdate – produces the current date and time in SQL Server's standard internal format for *datetime* values. *getdate* takes the NULL argument, ().

datename – produces the specified datepart (the first argument) of the specified date (the second argument) as a character string. Takes either a *datetime* or *smalldatetime* value as its second argument.

datepart – produces the specified date part (the first argument) of the specified date (the second argument) as an integer. Takes either a *datetime* or *smalldatetime* value as its second argument.

It is also used as an argument with **dateadd, datediff, datename**, and **datepart**. The following table lists the date parts, the abbreviations recognized by SQL Server, and the acceptable values.

Date Part	Abbreviation	Values
year	**yy**	1753 - 9999 (2079 for *smalldatetime)*
quarter	**qq**	1 - 4
month	**mm**	1 - 12
week	**wk**	1 - 54
day	**dd**	1 - 31
dayofyear	**dy**	1 - 366
weekday	**dw**	1 - 7 (Sun.-Sat.)
hour	**hh**	0 - 23
minute	**mi**	0 - 59
second	**ss**	0 - 59
millisecond	**ms**	0 - 999

Table 2-7: Date parts and their values

If the year is given with two digits, <50 is the next century (“25” is “2025”) and >=50 is this century (“50” is “1950”).

Milliseconds can be preceded either with a colon or a period. If preceded by a colon, the number means thousandths of a second. If preceded by a period, a single digit means tenths of a second, two digits mean hundredths of a second, and three digits mean thousandths of a second. For example, “12:30:20:1” means twenty and one-thousandth of a second past 12:30; “12:30:20.1” means twenty and one-tenth of a second past 12:30.

date – an argument used with **dateadd**, **datediff**, **datename**, and **datepart**. The date can be either the function **getdate**, a character string in one of the acceptable date formats (see “Datatypes”), an expression that evaluates to a valid date format, or the name of a *datetime* column.

datediff – calculates the number of date parts between two specified dates. It takes three arguments. The first is a date part. The second

and third are dates, either *datetime* or *smalldatetime* values. The result is a signed integer value equal to *date2 - date1*, in dateparts.

datediff results are always truncated, not rounded, when the result is not an even multiple of the datepart. For example, using **hour** as the datepart, the difference between "4:00 AM" and "5:50AM" is 1.

When you use **day** as the datepart, it counts the number of midnights between the two times specified. For example, the difference between January 1, 1992, 23:00 and January 2, 1992, 01:00 is 1; the difference between January 1, 1992 00:00 and January 1, 1992, 23:59 is 0.

The **month** datepart counts the number of first-of-the-months between two dates. For example, the difference between January 25 and February 2 is 1, while the difference between January 1 and January 31 is 0.

When you use the datepart **week** with **datediff**, you get the number of Sundays between the two dates, including the second date but not the first. For example, the number of weeks between Sunday, January 4 and Sunday, January 11 is 1.

If *smalldatetime* values are used, they are converted to *datetime* values internally for the calculation. Seconds and milliseconds in *smalldatetime* values are automatically set to 0 for the purpose of the difference calculation.

dateadd – adds an interval to a specified date. It takes three arguments—the datepart, a number, and a date. The result is a datetime value equal to the date plus the number of dateparts.

If the date argument is a *smalldatetime* value, the result is also a *smalldatetime*. You can use **dateadd** to add seconds or milliseconds to a *smalldatetime*, but it is only meaningful if the result date returned by **dateadd** changes by at least one minute.

Examples

1.

Command	Result
`select rightnow = getdate( )`	`Nov 25 1995 10:32AM`
`select datepart(month,getdate())`	`11`
`select datename(month getdate())`	`November`

This example assumes a current date of November 25, 1995, 10:32 a.m.

2. **select newpubdate = dateadd(day, 21, pubdate) from titles**

   Displays the new publication dates when the publication dates of all the books in the *titles* table slip 21 days.

3. **select newdate = datediff(day, pubdate, getdate()) from titles**

   This query finds the number of days elapsed between *pubdate* and the current date (obtained with the **getdate** function).

Comments

- Date functions can be used in the select list or **where** clause of a query.
- Use the *datetime* datatype only for dates after January 1, 1753. *datetime* values must be enclosed in single or double quotes. Use *char, nchar, varchar* or *nvarchar* for earlier dates. SQL Server recognizes a wide variety of date formats. See "**convert**" and "Datatypes" for more information.

  SQL Server automatically converts between character and *datetime* values when necessary (for example, when you compare a character value to a *datetime* value).
- The date part **weekday** or **dw** returns the day of the week (Sunday, Monday, etc.) when used with **datename**, and a corresponding number when used with **datepart**. The numbers that correspond to the names of weekdays depend on the **datefirst** setting. Some language defaults (including us_english) produce Sunday=1, Monday=2, and so on; others produce Monday=1, Tuesday=2,and so on.The default behavior can be changed on a per-session basis with **set datefirst.**

  Using **weekday** or **dw** with **dateadd** and **datediff** is not logical, and produces spurious results. Use **day** or **dd** instead.
- Since *smalldatetime* is accurate only to the minute, when a *smalldatetime* value is used with either **datename** or **datepart**, seconds and milliseconds are always zero.
- The **datediff** function produces results of datatype *int*, and causes errors if the result is greater than 2,147,483,647. For milliseconds, this is approximately 24 days, 20:31.846 hours. For seconds, this is 68 years, 19 days, 3:14:07 hours.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

See Also

Commands	**select**, **where** Clause
Functions	Datatype Conversion Functions
Topics	Datatypes, Expressions

Mathematical Functions

Function

Mathematical functions return values commonly needed for operations on mathematical data. Mathematical function names are not keywords.

Syntax

The mathematical functions take the general form:

```
function_name(arguments)
```

The chart below lists the types of arguments that are used in the mathematical functions.

Argument Type	Can Be Replaced By
approx_numeric	Any approximate numeric (*float*, *real*, or *double precision*) column name, variable, constant expression, or a combination of these.
integer	Any integer (*tinyint*, *smallint* or *int)* column name, variable, constant expression, or a combination of these.
numeric	Any exact numeric (*numeric*, *dec*, *decimal*, *tinyint*, *smallint*, or *int*), approximate numeric (*float*, *real*, or *double precision*), or *money* column, variable, constant expression, or a combination of these.
power	Any exact numeric, approximate numeric, or *money* column, variable, or constant expression, or a combination of these.

Table 2-8: Arguments used in mathematical functions

Each function also accepts arguments that can be implicitly converted to the specified type. For example, functions that accept approximate numeric types also accept integer types. SQL Server automatically converts the argument to the desired type.

If a function takes more than one expression of the same type, the expressions are numbered (for example, *approx_numeric1*, *approx_numeric2*).

Following are the mathematical functions, their arguments, and the results they return:

Function	Argument	Result
abs	(*numeric*)	Returns the absolute value of a given expression. Results are of the same type and have the same precision and scale as the numeric expression.
acos	(*approx_numeric*)	Returns the angle (in radians) whose cosine is the specified value**.**
asin	(*approx_numeric*)	Returns the angle (in radians) whose sine is the specified value**.**
atan	(*approx_numeric*)	Returns the angle (in radians) whose tangent is the specified value**.**
atn2	(*approx_numeric1, approx_numeric2*)	Returns the angle (in radians) whose tangent is (*approx_numeric1/approx_numeric2).*
ceiling	(*numeric*)	Returns the smallest integer greater than or equal to the specified value. Results are of the same type as the numeric expression. For *numeric* and *decimal* expressions, the results have a precision equal to that of the expression and a scale of 0.
cos	(*approx_numeric*)	Returns the cosine of the specified angle (in radians).
cot	(*approx_numeric*)	Returns the cotangent of the specified angle (in radians).
degrees	(*numeric*)	Converts radians to degrees. Results are of the same type as the numeric expression. For *numeric* and *decimal* expressions, the results have an internal precision of 77 and a scale equal to that of the expression. When *money* datatypes are used, internal conversion to *float* may cause loss of precision.
exp	(*approx_numeric*)	Returns the exponential value of the specified value.
floor	(*numeric*)	Returns the largest integer less than or equal to the specified value. Results are of the same type as the numeric expression. For *numeric* and *decimal* expressions, the results have a precision equal to that of the expression and a scale of 0.

Table 2-9: Mathematical functions

Function	Argument	Result
log	(*approx_numeric*)	Returns the natural logarithm of the specified value.
log10	(*approx_numeric*)	Returns the base 10 logarithm of the specified value.
pi	()	Returns the constant value of 3.1415926535897936.
power	(*numeric*, *power*)	Returns the value of *numeric* raised to the power *power*. Results are of the same type as *numeric*. For expressions of type *numeric* or *decimal*, the results have an internal precision of 77 and a scale equal to that of the expression.
radians	(*numeric*)	Converts degrees to radians. Results are of the same type as *numeric*. For expressions of type *numeric* or *decimal*, the results have an internal precision of 77 and a scale equal to that of the numeric expression. When money datatypes are used, internal conversion to *float* may cause loss of precision.
rand	([*integer*])	Returns a random *float* value between 0 and 1, using the optional *integer* as a seed value.
round	(*numeric*, *integer)*	Rounds the *numeric* so that it has *integer* significant digits. A positive integer determines the number of significant digits to the right of the decimal point; a negative *integer*, the number of significant digits to the left of the decimal point. Results are of the same type as the numeric expression and, for *numeric* and *decimal* expressions, have an internal precision of 77 and scale equal to that of the numeric expression.
sign	(*numeric*)	Returns the positive (+1), zero (0), or negative (-1). Results are of the same type, and have the same precision and scale, as the numeric expression.
sin	(*approx_numeric*)	Returns the sine of the specified angle (measured in radians).
sqrt	(*approx_numeric*)	Returns the square root of the specified value.
tan	(*approx_numeric*)	Returns the tangent of the specified angle (measured in radians).

Table 2-9: Mathematical functions (continued)

Examples

Statement	Result
`select floor(123)`	123
`select floor(123.45)`	123
`select floor(1.2345E2)`	123.000000
`select floor(-123.45)`	-124
`select floor(-1.2345E2)`	-124.000000
`select floor($123.45)`	123.00
`select ceiling(123.45)`	124
`select ceiling(-123.45)`	-123
`select ceiling(1.2345E2)`	124.000000
`select ceiling(-1.2345E2)`	-123.000000
`select ceiling($123.45)`	124.00
`select round(123.4545, 2)`	123.4500
`select round(123.45, -2)`	100.00
`select round(1.2345E2, 2)`	123.450000
`select round(1.2345E2, -2)`	100.000000

The **round** function always returns a value. If *integer* is negative and exceeds the number of significant digits in *numeric*, SQL Server returns a result of zero. (This is expressed in the form 0.00, where the number of zeros to the right of the decimal point is equal to the scale of *numeric*.) For example:

```
select round(55.55, -3)
```

returns a value of 0.00.

Comments

- Error traps are provided to handle domain or range errors of these functions. Users can **set** the **arithabort** and **arithignore** options to determine how domain errors are handled:
  - **arithabort arith_overflow** specifies behavior following a divide-by-zero error or a loss of precision. The default setting, **arithabort arith_overflow on**, rolls back the entire transaction or batch in which the error occurs. If you set **arithabort arith_overflow off**, SQL Server aborts the statement that causes the error but continues to process other statements in the transaction or batch.
  - **arithabort numeric_truncation** specifies behavior following a loss of scale by an exact numeric type during an implicit datatype conversion. (When an explicit conversion results in a loss of scale, the results are truncated without warning.) The default setting, **arithabort numeric_truncation on**, aborts the statement that causes the error but continues to process other statements in the

transaction or batch. If you set **arithabort numeric_truncation off**, SQL Server truncates the query results and continues processing.

- By default, the **arithignore arith_overflow** option is turned **off**, causing SQL Server to display a warning message after any query that results in numeric overflow. Set the **arithignore** option **on** to ignore overflow errors.

➤ ***Note***

The **arithabort** and **arithignore** options have been redefined for Release 10.0. If you use these options in your applications, examine them to be sure they still produce the desired effects.

- The **rand** function uses the output of a 32-bit pseudo-random integer generator. The integer is divided by the maximum 32-bit integer to give a double value between 0.0 and 1.0. The **rand** function is seeded randomly at server startup, so getting the same sequence of random numbers is unlikely, unless the user first initializes this function with a constant seed value. The **rand** function is a global resource. Multiple users calling the **rand** function progress along a single stream of pseudo-random values. If a repeatable series of random numbers is needed, the user must assure that the function is seeded with the same value initially and no other user calls **rand** while the repeatable sequence is desired.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

See Also

Commands	**set**
Functions	Datatype Conversion Functions, String Functions, text and image Functions

Row Aggregate Functions

Function

Generate summary values that appear as additional rows in the query results.

Syntax

```
Start of select statement

compute row_aggregate(column_name)
        [, row_aggregate(column_name)]...
   [by column_name [, column_name]...]
```

Keywords and Options

row_aggregate – is one of the following:

Name	Meaning
sum	Total of values in the (numeric) column
avg	Average of values in the (numeric) column
min	Lowest value in the column
max	Highest value in the column
count	Number of non-null values in the column

Table 2-10: Row aggregate functions

column_name – is the name of a column. It must be enclosed in parentheses. Only exact numeric, approximate numeric, and money columns can be used with **sum** and **avg**.

One **compute** clause can apply the same function to several columns. When using more than one function, use more than one **compute** clause.

by – indicates that row aggregate values are to be calculated for subgroups. Whenever the value of the **by** item changes, row aggregate values are generated. If you use **by**, you must use **order by**.

Listing more than one item after **by** breaks a group into subgroups and applies a function at each level of grouping.

Examples

```
select type, price
from titles
where price > $10 and type like "%cook"
order by type, price
compute sum(price) by type
```

```
 type       price
---------- ----------
mod_cook        19.99
           sum
           ----------
                19.99
 type      price
---------- ----------
trad_cook       11.95
trad_cook       14.99
trad_cook       20.95
           sum
           ----------
                47.89

(6 rows affected)
```

Calculates the sum of the price of each type of cookbook over $10. (See "**compute** Clause" for more examples and information.)

Comments

- The row aggregates make it possible to retrieve detail and summary rows with one command. The aggregate functions, on the other hand, ordinarily produce a single value for all the selected rows in the table or for each group, and these summary values are shown as new columns.

  The following examples illustrate the differences:

```
select type, sum(price), sum(advance)
from titles
where type like "%cook"
group by type
```

```
type
----------  ----------  ----------------
mod_cook         22.98         15,000.00
trad_cook        47.89         19,000.00

(2 rows affected)
```

```
select type, price, advance
from titles
where type like "%cook"
order by type
compute sum(price), sum(advance) by type
```

```
type        price       advance
----------  ----------  ----------------
mod_cook          2.99         15,000.00
mod_cook         19.99              0.00
            sum         sum
            ----------  ----------------
                 22.98         15,000.00
type         price         advance
----------  ----------  ----------------
trad_cook        11.95          4,000.00
trad_cook        14.99          8,000.00
trad_cook        20.95          7,000.00
            sum         sum
            ----------  ----------------
                 47.89         19,000.00
(7 rows affected)
```

- The columns in the **compute** clause must appear in the select list.
- If the **ansinull** option is set **off** (the default), there is no warning when a row aggregate encounters a null. If **ansinull** is set **on**, a query returns the following SQLSTATE warning when a row aggregate encounters a null:

  ```
  Warning- null value eliminated in set function
  ```

- You cannot use **select into** in the same statement as a **compute** clause because statements that include **compute** generate tables that include the summary results, which are not stored in the database.
- When you sum or average integer data, SQL Server treats the result as an *int* value, even if the datatype of the column is *smallint* or *tinyint*. To avoid overflow errors in DB-Library programs, make all variable declarations for results of averages or sums type *int*.
- In a **select** statement with a **compute** clause, the order of columns in the select list overrides the order of the aggregates in the **compute** clause. For example:

  ```
  select a, b, c
  from test
  compute sum(c), max(b), min(a)
  ```

```
a           b       c
--------- ------- -------
        1       2       3
        3       2       1
                  sum
                  =======
                        4
          max
          =======
                2
min
=========
        1
(3 rows affected)
```

DB-Library programmers must be aware of this in order to put the aggregate results in the right place.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

See Also

Commands	**compute** Clause, **group by** and **having** Clauses, **order by** Clause, **select**
Functions	Aggregate Functions

String Functions

Function

Operate on binary data, character strings, and expressions. String functions can be nested, and they can be used anywhere an expression is allowed. When you use constants with a string function, enclose them in single or double quotes. String function names are not keywords.

Syntax

The string functions take the general form:

***function_name*(*arguments*)**

The chart below lists the arguments used in string functions. If a function takes more than one argument of the same type, the arguments are numbered (for example, *char_expr1*, *char_expr2)*.

Argument Type	Can Be Replaced By
char_expr	A character-type column name, variable, or constant expression of *char, varchar, nchar* or *nvarchar* type. Functions that also accept *text*-type column names are noted in the explanation. Constant expressions must be enclosed in quotation marks.
expression	A binary or character column name, variable or constant expression. Can be *char, varchar, nchar* or *nvarchar* data (as for *char_expr,* above) plus *binary* or *varbinary.*
pattern	A character expression (of *char* or *varchar* datatype) that may include any of the pattern-match wildcards supported by SQL Server. Only the **patindex** function accepts *pattern* (and therefore, wildcards).
approx_numeric	Any approximate numeric (*float*, *real*, or *double precision*) column name, variable, or constant expression.
integer_expr	Any integer (*tinyint, smallint* or *int)* column name, variable, or constant expression. Maximum size ranges are noted, as they apply.
start	An *integer_expr.*
length	An *integer_expr.*

Table 2-11: Arguments used in string functions

Each function also accepts arguments that can be implicitly converted to the specified type. For example, functions that accept approximate numeric expressions also accept integer expressions. SQL Server automatically converts the argument to the desired type.

The following table lists function names, arguments, and results.

Function	Argument	Result
ascii	(*char_expr*)	Returns the ASCII code for the first character in the expression.
char	(*integer_expr)*	Converts a single-byte *integer* value to a *character* value. (**char** is usually used as the inverse of **ascii**.) *integer_expr* must be between 0 and 255. Returns a *char* datatype. If the resulting value is the first byte of a multibyte character, the character may be undefined.
charindex	(*expression1*, *expression2)*	Searches *expression2* for the first occurrence of *expression1* and returns an integer representing its starting position. If *expression1* is not found, returns 0. If *expression1* contains wildcard characters, **charindex** treats them as literals.
char_length	(*char_expr)*	Returns an integer representing the number of characters in a character expression or *text* value. For variable-length data, **char_length** strips the expression of trailing blanks before counting the number of characters. For multi-byte character sets, the number of characters in the expression is usually less than the number of bytes; use **datalength** (See "System Functions") to determine the number of bytes.
difference	(*char_expr1*, *char_expr2)*	Returns an integer representing the difference between two **soundex** values. See **soundex**, below.
lower	(*char_expr)*	Converts upper to lower case, returning a character value.
ltrim	(*char_expr)*	Removes leading blanks from the character expression. Only values equivalent to the space character in the current character set are removed.

Table 2-12: Function names, arguments, and results

Function	Argument	Result
patindex	("%*pattern*%", *char_expr* [, **using** {**bytes** | **chars** | **characters**}])	Returns an integer representing the starting position of the first occurrence of *pattern* in the specified character expression, or a zero if *pattern* is not found. By default, **patindex** returns the offset in characters; to return the offset in bytes (multibyte character strings), specify **using bytes**. The '%' wildcard character must precede and follow *pattern* (except when searching for first or last characters). See "Wildcards" for a description of the wildcard characters that can be used in *pattern*. Can be used on *text* data.
replicate	(*char_expr, integer_expr)*	Returns a string with the same datatype as *char_expr*, containing the same expression repeated the specified number of times or as many times as will fit into a 255 byte space, whichever is less.
reverse	(*char_expr)*	Returns the reverse of *char_expr*; if *char_expr* is "abcd", it returns "dcba".
right	(*char_expr, integer_expr)*	Returns the part of the character expression starting the specified number of *characters* from the right. Return value has the same datatype as the character expression.
rtrim	(*char_expr)*	Removes trailing blanks. Only values equivalent to the space character in the current character set are removed.
soundex	(*char_expr)*	Returns a four-character **soundex** code for character strings that are composed of a contiguous sequence of valid single- or double-byte roman letters.
space	(*integer_expr)*	Returns a string with the indicated number of single-byte spaces.

Table 2-12: Function names, arguments, and results (continued)

Function	Argument	Result
str	(*approx_numeric* [, *length* [, *decimal*]])	Returns a character representation of the floating point number. *length* sets the number of characters to be returned (including the decimal point, all digits to the right and left of the decimal point, and blanks); *decimal* sets the number of decimal digits to be returned. *length* and *decimal* are optional. If given, they must be non-negative. Default *length* is 10; default *decimal* is 0. **str()** rounds the decimal portion of the number so that the results fit within the specified *length*.
stuff	(*char_expr1*, *start*, *length*, *char_expr2*)	Deletes *length* characters from *char_expr1* at *start*, then inserts *char_expr2* into *char_expr1* at *start*. To delete characters without inserting other characters, *char_expr2* should be NULL (not "", which indicates a single space).
substring	(*expression*, *start*, *length*)	Returns part of a character or binary string. *start* specifies the character position at which the substring begins. *length* specifies the number of characters in the substring.
upper	(*char_expr*)	Converts lower to upper case, returning a character value.

Table 2-12: Function names, arguments, and results (continued)

Examples

1.
```
select au_lname, substring(au_fname, 1, 1)
from authors
```

   Displays the last name and first initial of each author, for example, *Bennet A*.

2.
```
select (au_lname + ",  "
    + substring(au_fname, 1, 1) + ".")
from authors
```

   Displays the last name followed by a comma, and the first initial with a period of each author, for example, *Bennet, A*.

3. **select substring ((pub_id + title_id), 1, 6)**
   from titles

   Displays the first six characters of the concatenation of each *pub_id* and *title_id*. In other words, displays all four characters of the *pub_id* and the first two characters of the *title_id*.

4. **select charindex("wonderful", notes)**
   from titles
   where title_id = "TC3218"

   Returns the position at which the character expression "wonderful" begins in the *notes* column of *titles*.

5. **select au_id, patindex("%circus%", copy)**
   from blurbs

   Selects the author ID and the starting character position of the words "circus" in the *copy* column. It returns "0" if the pattern is not found. **patindex** can be used on all character data, including *text* data, and the "%" wildcard characters must be used unless searching for the first of last characters of a field.

6. Below are some of the results of the **right**, **str**, and **stuff** functions:

Statement	Results
select right("abcde", 3)	cde
select right("abcde", 6)	abcde
select str(1234.7, 4)	1235
select stuff("abc", 2, 1,"xyz")	axyzc

Table 2-13: Results of the right, str, and stuff functions

Comments

- **patindex** can be used on *char*, *varchar nchar*, *nvarchar* and *text* data. The other string functions can only be used on *char*, *varchar*, *binary*, and *varbinary* datatypes, plus those datatypes which convert implicitly to *char* or *varchar*.
- The string functions can be nested. See examples 3 and 4.
- When you use constants with the string functions, enclose them in single or double quotes.
- The string functions can be used in a select list, in a **where** clause, and anywhere an expression is allowed.
- Length and decimal arguments to **str** (if supplied) must be positive. The default length is 10. The default decimal is 0. The length should be long enough to accommodate the decimal point and the number's sign. The decimal portion of the result is

rounded to fit within the specified length. If the integer portion of the number does not fit within the length, however, **str** returns a row of asterisks of the specified length. For example:

```
select str(123.456, 2, 4)

--
**
(1 row affected)
```

A short *approx_numer* is right justified in the specified length, and a long *approx_numer* is truncated to the specified number of decimal places.

- The **stuff** function inserts a string into another string. It deletes a specified length of characters in *expr1* at the start position. It then inserts *expr2* string into *expr1* string at the start position.

  If the start position or the length is negative, a NULL string is returned. If the start position is longer than *expr1*, a NULL string is returned. If the length to delete is longer than *expr1*, *expr1* is deleted through its last character.

```
select stuff("abc", 2, 3, "xyz")

----
axyz
(1 row affected)
```

  To use **stuff** to delete a character, replace *expr2* with NULL, not with empty quotation marks. Using " " to specify a null character replaces it with a space.

```
select stuff("abcdef", 2, 3, null)

go
---
aef
(1 row affected)


select stuff("abcdef", 2, 3, "")

----
a ef
(1 row affected)
```

- The **soundex** function converts an alpha string to a four-digit code for use in locating similar-sounding words or names. All vowels

are ignored unless they constitute the first letter of the string. For example, these two strings return the same value:

```
select soundex ("smith"), soundex ("smythe")

----- -----
S530  S530
```

- The **difference** function compares two strings and evaluates the similarity between them, returning a value from 0 to 4. 4 is the best match.

```
select difference("smithers", "smothers")

---------
4
(1 row affected)
select difference("smothers", "brothers")

---------
2
(1 row affected)
```

  The string values must be composed of a contiguous sequence of valid single- or double-byte roman letters.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

See Also

Commands	select
Functions	Datatype Conversion Functions, Mathematical Functions, text and image Functions

System Functions

Function

Return special information from the database.

Syntax

System functions take the general form:

***function_name*(*arguments*)**

The chart below lists types of arguments usually preceded by **select**.

Argument Type	Can Be Replaced By
character_ expression	A column name, variable, constant expression, or a combination of any of these that evaluates to a single value of type *char* or *varchar*.
column_id	A numeric expression that is a column ID of a column. These are stored in the *colid* column of *syscolumns*.
database_id	A numeric expression that is the ID for a database. These are stored in the *db_id* column of *sysdatabases*.
doampg	The *sysindexes* pointer to the data pages' object allocation map.
expression	A column name, variable, constant expression, or a combination of any of these that evaluates to a single value. It can be of any datatype.
ioampg	The *sysindexes* pointer to the index pages' object allocation map.
object_id	A numeric expression that is an object ID for a table, view, or other database object. These are stored in the *id* column of *sysobjects*.
object_name	A character expression that is the name of a database object, such as a table, view, procedure, trigger, default, or rule. The name can be fully qualified (that is, it can include the database and owner name). It must be enclosed in quotes.

Table 2-14: Arguments used in system functions

Some character expression arguments, such as *server_user_name*, can include variables or literals, or concatenated literals, column names,

and variables. All literals must be enclosed in quotation marks, as shown in the examples that follow the chart.

Function names, arguments, and results are listed in the following table. Where the argument is optional, the function returns the current value.

Function	Argument	Result
col_name	(*object_id*, *column_id* [, *database_id*])	Returns the column name.
col_length	(*object_name*, *column_name*)	Returns the defined length of column. Use **datalength** to see the actual data size.
curunreservedpgs	(*database_id*, *page_number*, *free_pages*)	Returns the number of free pages in the disk piece that contains *page_number*. Use *free_pages* to specify a default value to be returned by the function. If the database is open, **curunreservedpgs** replaces this value with the actual number of free pages stored in memory for that disk piece.
data_pgs	(*object_id*, {*doampg* | *ioampg*})	Returns the number of pages used by table (*doampg)* or index (*ioampg*). The result does not include pages used for internal structures.
datalength	(*expression*)	Returns the length of *expression* in bytes. *expression* is usually a column name. If *expression* is a character constant, it must be enclosed in quotes.
db_id	([*database_name*])	Returns the database ID number. *database_name* must be a character expression; if it is a constant expression, it must be enclosed in quotes. If no *database_name* is supplied, **db_id** returns the ID number of the current database.

Table 2-15: System functions, arguments, and results

Function	Argument	Result
db_name	([*database_id*])	Returns the database name. *database_id* must be a numeric expression. If no *database_id* is supplied, **db_name** returns the name of the current database.
host_id	( )	Returns the host process ID of the client process (not the Server process).
host_name	( )	Returns the current host computer name of the client process (not the Server process).
index_col	(*object_name*, *index_id*, *key_#* [, *user_id*])	Returns the name of the indexed column; returns NULL if *object_name* is not a table or view name.
isnull	(*expression1*, *expression2*)	Substitutes the value specified in *expression2* when *expression1* evaluates to NULL.The datatypes of the expressions must convert implicitly, or you must use the **convert** function.
lct_admin	({{ "**lastchance**" | "**logfull**" | "**unsuspend**"}, *database_id*} | "**reserve**", *log_pages*})	Manages the log segment's last-chance threshold. **lastchance** creates a last-chance threshold in the specified database. **logfull** returns 1 if the last-chance threshold has been crossed in the specified database and 0 if it has not. **unsuspend** awakens suspended tasks in the database and disables the last-chance threshold if that threshold has been crossed. **reserve** returns the number of free log pages required to successfully dump a transaction log of the specified size.
object_id	(*object_name*)	Returns the object ID.
object_name	(*object_id*[, *database_id*])	Returns the object name.

Table 2-15: System functions, arguments, and results (continued)

Function	Argument	Result
proc_role	(**"sa_role"** | **"sso_role"** | **"oper_role"**)	Checks to see if the invoking user possesses the correct role to execute the procedure. Returns 1 if the invoker has the required role. Otherwise, returns 0.
reserved_pgs	(*object_id*, {*doampg* | *ioampg*})	Returns the number of pages allocated to table or index. This function **does** report pages used for internal structures.
rowcnt	(*doampg*)	Returns the number of rows in a table (estimate).
show_role	( )	Returns the user's current active roles, if any (**sa_role**, **sso_role**, or **oper_role**). If the user has no roles, returns NULL.
suser_id	([*server_user_name*])	Returns the server user's ID number from *syslogins*. If no *server_user_name* is supplied, it returns the server ID of the current user.
suser_name	([*server_user_id*])	Returns the server user's name. Server user's ID's are stored in *syslogins*. If no *server_user_id* is supplied, it returns the name of the current user.
used_pgs	(*object_id, doampg, ioampg*)	Returns the total number of pages used by a table and its clustered index.
tsequal	(*timestamp, timestamp2*)	Compares *timestamp* values to prevent update on a row that has been modified since it was selected for browsing. *timestamp* is the timestamp of the browsed row; *timestamp2* is the timestamp of the stored row. Allows you to use browse mode without calling DB-Library. (See "Browse Mode.")
user		Returns the user's name.

Table 2-15: System functions, arguments, and results (continued)

Function	Argument	Result
user_id	([*user_name*])	Returns the user's ID number. Reports the number from *sysusers* in the current database. If no *user_name* is supplied, it returns the ID of the current user.
user_name	([*user_id*])	Returns the user's name, based on the user's ID in the current database. If no *user_id* is supplied, it returns the name of the current user.
valid_name	(*character_expression*)	Returns 0 if *character_expression* is not a valid identifier (illegal characters or *string* is more than 30 bytes long), a nonzero number if *character_expression* is a valid identifier.
valid_user	(*server_user_id*)	Returns 1 if the specified ID is a valid user or alias in at least one database on this SQL Server. You must have the **sa_role** or **sso_role** role to use this function on a *server_user_id* other than your own.

Table 2-15: System functions, arguments, and results (continued)

Examples

1. **`select user_id("harold")`**

   Returns Harold's ID, 13.

2. **`select user_name(13)`**

   Returns the login name for user 13, which is *harold.*

3. **`select db_name()`**

   Returns the name of the current database.

4. **`select avg(isnull(price,$10.00))`**
   `from titles`

   Finds the average of the prices of all titles, substituting the value $10.00 for all NULL entries in *price.*

5.
```
select isnull(price,0)
from titles
```

   Returns all rows from *titles*, replacing null values in *price* with "0".

6.
```
select x = col_length("titles", "title")
```

   Finds the length of the *title* column in the *titles* table. The "x" is included to give a column heading to the result.

7.
```
select Length = datalength(pub_name)
from publishers
```

   Finds the length of the *pub_name* column in the *publishers* table.

8.
```
select name from sysusers
where name = user_name(1)
```

   Finds all rows in *sysusers* where the name is equal to the result of applying the system function **user_name** to user ID #1.

9.
```
select object_id("pubs2." + user_name() + ".mytab")
```

   Finds the object ID for the table *mytab* owned by the current user.

10.
```
select sysobjects.name,
Pages = data_pgs(sysindexes.id, doampg)
from sysindexes, sysobjects
where sysindexes.id = sysobjects.id
    and sysindexes.id > 100
    and (indid = 1 or indid = 0)
```

   Estimates the number of data pages used by user tables (which have objects IDs greater than 100). An *indid* of 0 indicates a table without a clustered index; an *indid* of 1 indicates a table with a clustered index. This query does not return rows for any nonclustered indexes or include the size of text chains (which have *indid = 255)*. To see this information, use *ioampg* in the **data_pgs** function, and *indid* > 1 in the **where** clause.

11.
```
select show_role()
```

   Returns a list of your current active roles.

12.
```
select proc_role("sso_role")
```

   Checks to see whether the user has the role of System Security Officer. Returns "1" if so, otherwise "0".

Comments

- All system functions except **user** require parentheses.

- When the argument to a system function is optional, the current database, host computer, server user, or database user is assumed.
- The system functions can be used in a **select** list, in a **where** clause, and anywhere an expression is allowed.
- The server user ID of the "sa" account is 1. When other users are granted the role of System Administrator, they retain their own server user IDs.
- Server user ID -1 is always reserved for a guest account.
- The user ID of the Database Owner is always 1. When a System Administrator uses a database, he or she becomes Database Owner, with user ID 1. If the user turns off his or her System Administrator role for a session, the user assumes his or her own database user ID. If the user has no database user ID in the current database, he or she cannot execute **set role "sa_role" off.**
- The **user_name** function returns the name of the current user if the argument is less than 0.
- **col_length** finds the defined length of a column; **datalength** finds the actual length of the data stored in each row. **datalength** is useful on *varchar*, *varbinary*, *text* and *image* datatypes, since these datatypes can store variable lengths. **datalength** of any NULL data returns NULL. For all other datatypes, **datalength** reports their defined length. For *text* and *image* columns, **col_length** returns 16, the length of the *binary(16)* pointer to the actual text page.
- **data_pgs** returns the number of pages used by a table or index; it does **not** include pages used for internal structures. **used_pgs** returns the number of pages used by a table and its clustered index; it **does** include pages used for internal structures.
- To check the size of the transaction log, the table *syslogs*, use:

  ```
  select reserved_pgs(id, doampg)
  ```

- Use **proc_role** within a stored procedure to guarantee that only users with a specific role can execute it. While **grant execute** can also restrict execute permission on a stored procedure, users without the required role can still be granted permission to execute it. Only **proc_role** provides a fail-safe way to prevent inappropriate access to a particular stored procedure.
- **rowcnt** estimates the number of rows, based on the average number of rows per page in a table, and the number of pages in the tablet. This value can vary unexpectedly when the server

reboots and transactions have to be recovered. Correct values are restored by running **update statistics, dbcc checktable** or **dbcc checkdb.**

Standards and Compliance

Standard	Compliance Level
SQL89	The **user** function is compliant. All other system functions are Transact-SQL extensions.
SQL92	The **user** function is entry level compliant. All other system functions are Transact-SQL extensions.

See Also

Commands	**grant, select**

text and *image* Functions

Function

Operate on *text* and *image* data. Text and image built-in function names are not keywords. Use the **set textsize** option to limit the amount of *text* or *image* data that a **select** statement retrieves.

Syntax

Text and image functions take the general form:

***function_name*(*argument*)**

Function names, arguments, and results are listed in the following table. A **pattern** is an alphanumeric *expression* of *char*, *varchar*, *binary* or *varbinary* type used for pattern matching.

Function	Argument	Result
patindex	("%*pattern*%", *char_expr* [, **using** {**bytes** | **chars** | **characters**}])	Returns an integer value representing the starting position of the first occurrence of *pattern* in the specified character expression, or zero if *pattern* is not found. By default, **patindex** returns the offset in characters; to return the offset in bytes for multibyte character strings, specify **using bytes**. The % wildcard character must precede and follow *pattern*, except when searching for first or last characters. See "Wildcards" for a description of the wildcard characters that can be used in *pattern*.
textptr	(*text_columname*)	Returns the text pointer value, a 16-byte binary value. The text pointer is checked to ensure that it points to the first text page.
textvalid	("*table_name.col_name*", *textpointer*)	Checks if a given text pointer is valid. Note that the identifier for a *text* or *image* column must include the table name. Returns 1 if the pointer is valid, or 0 if the pointer is invalid.

Table 2-16: text and image functions, arguments, and results

Examples

1.
```
declare @val varbinary(16)
select @val = textptr(blurb) from texttest
where title_id = "BU7832"
readtext texttest.blurb @val 1 5
```

   This example uses the **textptr** function to locate the *text* column, *blurb*, associated with *title_id* BU7832 in table *texttest*. The text pointer is put into a local variable *@val* and supplied as a parameter to the **readtext** command, which returns 5 bytes starting at the second byte (offset of 1).

2.
```
select title_id, textptr(blurb) from texttest
```

   Selects the *title_id* column and the 16-byte text pointer of the *blurb* column from the *texttest* table.

3.
```
select textvalid ("texttest.blurb",
textptr(blurb)) from texttest
```

   Reports whether a valid text pointer exists for each value in the *blurb* column of the *texttest* table.

Comments

- If a *text* or *image* column has not been initialized by a non-null **insert** or by any **update** statement, *textptr* returns a NULL pointer. Use **textvalid** to check whether a text pointer exists. You cannot use **writetext** or **readtext** without a valid text pointer.
- Use the **datalength** function to get the length of data in *text* and *image* columns. See "System Functions" for details.
- The **patindex** function returns the character position of a given character string in *text* columns. See also "String Functions."
- *text* and *image* columns cannot be used:
  - As parameters to stored procedures.
  - As values passed to stored procedures.
  - As local variables.
  - In **order by**, **compute**, and **group by** clauses.
  - In an index.
  - In a **where** clause, except with the keyword **like**.
  - In joins.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

See Also

Commands	**readtext, set, writetext**
Functions	Datatype Conversion Functions, Mathematical Functions, String Functions, System Functions
Topics	text and image Datatypes

Topics

3 Transact-SQL Topics

This chapter presents general Transact-SQL reference information arranged alphabetically, by topic. Each topic describes the commands, system procedures, and functions that provide a particular type of functionality. *Table 3-1: Transact-SQL topics* lists the topics discussed in this chapter.

Topic	Description
Auditing	Records security-related system activity in an audit trail that can be used to detect penetration of the system and misuse of system resources.
Batch Queries	Combine a set of SQL statements that are submitted together and executed as a group, one after the other.
Browse Mode	Allows front end applications using Open Client and a host programming language to perform updates while viewing data.
Comments	Attach comments to SQL statements, statement blocks, and system procedures.
Control-of-Flow Language	Controls the flow of execution of SQL statements, statement blocks, and stored procedures.
Cursors	Provide access to the set of rows returned by a SQL query.
Datatypes	Specify the type of information, size, and storage format of columns, stored procedure parameters, and local variables.
Disk Mirroring	Creates a software mirror of a user database device, the master device, or a database device used for user database transaction logs.
Expressions	Combine constants, literals, functions, column identifiers, variables, and operators.
Identifiers	Name database objects such as databases, tables, views, columns, indexes, triggers, procedures, defaults, rules, and cursors.
IDENTITY Columns	Uniquely identify each row within a table. Each time you insert a row into the table, SQL Server automatically provides a unique, sequential value for the IDENTITY column.

Table 3-1: Transact-SQL topics

Topic	Description
Joins	Compare two or more tables (or views) by specifying a column from each, comparing the values in those columns row by row, and concatenating rows that have matching values.
Login Management	Verifies the identities of SQL Server users and administers SQL Server login accounts.
Null Values	Marks columns having an unknown value (as opposed to those that have 0 or blank as a value).
Parameters	Are arguments to a stored procedure.
Roles	Provide greater individual accountability for users performing system administration and security-related tasks on SQL Server.
Search Conditions	Set the conditions in a **where** or **having** clause.
Subqueries	Nest a **select** statement inside a **select**, **insert**, **update**, or **delete** statement, another subquery, or anywhere an expression is allowed (if it returns a single value).
Temporary Tables	Store data in *tempdb* for the duration of the current session.
text and *image* Datatypes	Store up to 2,147,483,647 (2^{31} - 1) bytes of printable characters *(text*) or Sybase binary data (*image*) in a linked list of data pages that are external to the table.
Transactions	Group SQL statements so that they are treated as a unit: either all statements in the group are executed, or no statements in the group are executed.
Variables (Local and Global)	Assume values assigned by a **declare** statement (local variables) or supplied by SQL Server (global variables).
Wildcard Characters	Represent one or more characters in a string. Wildcards are used with the keyword **like** to find character and date strings that match a particular pattern.

Table 3-1: Transact-SQL topics (continued)

Auditing

Function

Record security-related system activity in an audit trail that can be used to detect penetration of the system and misuse of system resources.

The Audit System

The audit system consists of the *sybsecurity* database and a set of stored procedures that allow you to selectively set the audit options you need.

You can choose to audit the following:

- Within a server you can audit logins and logouts, server boots, RPC connections made from other servers, errors, and execution of commands requiring special roles.
- At the database level, you can audit the use of the **grant, revoke, truncate table,** and **drop** commands within a database; the use of the **drop** and **use** commands on a database; and references to a specific database from within another database.
- At the user level, you can audit a specified user's attempts to access tables and views, and you can audit the text of commands submitted to the server by a user.
- At the object level, you can audit accesses to specified tables and views and execution of stored procedures and triggers.

Auditing is discussed in more detail in the *System Administration Guide*.

The *sybsecurity* Database

The *sybsecurity* database consists of:

- The *sysaudits* table, which is the audit trail. All audit records are written into *sysaudits*.
- The *sysauditoptions* table, which contains one row for each global audit option.
- All of the other default system tables, derived from the *model* database.

See Appendix B, "The System Tables," for a description of each system table.

Installing the Audit System

The audit system and *sybsecurity* database can be installed at any time using the **sybinit** installation program. See the *SQL Server Installation Guide* for your platform for information.

Setting Audit Options

System Security Officers can determine the type of auditing to be performed in the system. Auditing options are managed using the following system procedures:

System Procedure	Description
sp_auditoption	Enables and disables system-wide auditing and global audit options
sp_auditdatabase	Establishes auditing of different types of events within a database, or of references to objects within that database from another database
sp_auditobject	Establishes selective auditing of accesses to tables and views
sp_auditsproc	Audits the execution of stored procedures and triggers
sp_auditlogin	Audits a user's attempts to access tables and views, or the text of commands that the user executes
sp_addauditrecord	Allows users to enter user-defined audit records (comments) into the audit trail

Table 3-2: System procedures used to manage auditing options

The Audit Queue

When an audited event occurs, an audit record first goes to the audit queue, where it waits until it can be added to the audit trail. You can configure the size of the audit queue with the **audit queue size** option to **sp_configure**. The size of the audit queue is set according to your needs. See ''audit queue size'' in the *System Administration Guide* for information on how to configure the size of the audit queue and effects of different queue sizes.

The Audit Trail

The audit trail is contained in the *sybsecurity..sysaudits* table. It is a special table in that the only operations allowed on it are **select** and **truncate table**, and these operations can be performed only by System Security Officers. The columns of *sysaudits* are described in

Appendix B, "The System Tables." Procedures for archiving the audit trail are discussed in the *System Administration Guide.*

Permissions

Only System Security Officers can enable auditing and execute the system procedures to set audit options. The exception is **sp_addauditrecord**, which can be run by any user who has been granted permission to execute it.

See Also

System procedures	**sp_addauditrecord, sp_auditdatabase, sp_auditinstall, sp_auditlogin, sp_auditobject, sp_auditoption, sp_auditsproc**

Batch Queries

Function

A batch or batch file is a set of SQL statements submitted together and executed as a group, one after the other. A batch is terminated by an end-of-batch signal. With the **isql** utility, this is the word "go" on a line by itself.

Following is an example of a batch that contains two SQL statements:

```
select count(*) from titles
select count(*) from authors
go
```

Submitting Batches to *isql*

You can submit batches to **isql** either interactively or from an operating system file. A file submitted to **isql** can include more than one batch of SQL statements if each batch is terminated by the word "go". For details on the **isql** utility, see the *SQL Server Utility Programs* for your platform.

Referencing Database Objects

Before referencing objects in a database, you must issue a **use** statement for that database. For example:

```
use master
go
select count(*)
from sysdatabases
go
```

Creating Objects Within a Batch

Certain data definition statements cannot be combined with other statements within a batch. These include **create procedure**, **create rule**, **create default**, **create trigger** and **create view.**

create database, **create table**, and **create index can** be combined with other statements in a single batch. The following batch creates the table *test*, inserts a row into the table, and then retrieves the new row:

```
create table test
  (column1 char(10), column2 int)
insert test
  values ("hello", 598)
select * from test
go
```

Dropping Objects Within a Batch

You cannot **drop** an object and then reference or recreate it in the same batch.

Applying Rules and Defaults Within a Batch

Rules and defaults cannot be bound to columns and used during the same batch. **sp_bindrule** or **sp_bindefault** cannot be in the same batch as **insert** statements that invoke the rule or default.

Setting Options in a Batch

Some options set with a **set** statement do not take effect until the end of the batch. See "**set**" for details.

See Also

Commands	**create database, create default, create index, create procedure, create rule, create table, create trigger, create view, set**

Browse Mode

Function

Browse mode supports the ability to perform updates while viewing data. It is used in front end applications using DB-Library and a host programming language.

Browsing a Table

To browse a table in a front-end application, append the **for browse** keywords to the end of the **select** statement sent to SQL Server. A **for browse** clause cannot be used in statements involving the **union** operator or in cursor declarations. The use of the keyword **holdlock** is forbidden in a **select** statement that includes the **for browse** option.

For example:

[*Start of* **select** *statement in an Open Client application*]

```
for browse
```

[**for browse** *is the last clause in a* **select** *statement. See* select.]

[*more application*]

A table can be browsed in a front-end application if its rows have been timestamped.

Timestamping a New Table for Browsing

When creating a new table for browsing, include a column named *timestamp* in the table definition. The column is automatically assigned a datatype of *timestamp*; you do not have to specify its datatype. For example:

```
create table newtable(col1 int, timestamp,
    col3 char(7))
```

Whenever you insert or update a row, SQL Server timestamps it by automatically assigning a unique *varbinary* value to the *timestamp* column.

Timestamping an Existing Table

To prepare an existing table for browsing, add a column named *timestamp* with **alter table**. For example:

```
alter table oldtable add timestamp
```

A *timestamp* column with a NULL value is added to each existing row. To generate a timestamp, update each existing row without specifying new column values.

For example:

```
update oldtable
set col1 = col1
```

Comparing *timestamp* Values

Use the **tsequal** system function to compare timestamps when you are using browse mode in a front-end application. For example, the following statement updates a row in *publishers* that has been browsed. It compares the *timestamp* column in the browsed version of the row with the hexadecimal timestamp in the stored version. If the two timestamps are not equal, you receive an error message and the row is not updated.

```
update publishers
set city = “Springfield”
where pub_id = “0736”
and tsequal(timestamp,0x0001000000002ea8)
```

Standards and Compliance

for browse is a Transact-SQL extension.

See Also

Functions	System Functions
Topics	Datatypes
System procedures	**sp_primarykey**

Comments

Function

Comments attach explanatory text to SQL statements, statement blocks, and system procedures. Comments are not executed.

A comment can be inserted on a line by itself or at the end of a command line. Two comment styles are available: the "slash-asterisk" style:

```
/* text of comment (slash-asterisk style) */
```

and the "double-hyphen" style:

```
-- text of comment (double-hyphen style)
```

Slash-Asterisk Style Comments

Multiple-line comments are allowed, as long as they are surrounded by "/*" and "*/".

A stylistic convention often used for multiple-line comments is to begin the first line with "/*" and subsequent lines with "**". The comment ends with "*/" as usual.

The "/*" form permits nesting.

Following are examples:

```
/* this procedure finds rules by user name*/
create procedure findmyrule @nm varchar(30) = null
as
if @nm is null
begin
  print "You must give a user name"
  return
  print "I have returned"
/* this statement follows return,
** so won't be executed */
end
else      /* print the rule names and IDs, and
          the user ID */
  select sysobjects.name, sysobjects.id,
     sysobjects.uid
  from sysobjects, master..syslogins
  where  master..syslogins.name = @nm
  and sysobjects.uid = master..syslogins.suid
  and sysobjects.type = "R"
```

Double-Hyphen Style Comments

This comment style begins with two consecutive hyphens followed by a space, and terminates with a newline character. Multiple-line comments are therefore not possible.

SQL Server will not interpret two consecutive hyphens within a string literal or within a "/*"-style comment as signalling the beginning of a comment.

To represent an expression containing two minuses in sequence (binary followed by unary), put a space or open parenthesis between the two hyphens.

Following are examples:

```
-- this procedure finds rules by user name
create procedure findmyrule @nm varchar(30) = null
as
if @nm is null
begin
  print "You must give a user name"
  return
  print "I have returned"
-- each line of a multiple-line comment
-- must be marked separately.
end
else    -- print the rule names and IDs, and
        -- the user ID
  select sysobjects.name, sysobjects.id,
     sysobjects.uid
  from sysobjects, master..syslogins
  where  master..syslogins.name = @nm
  and sysobjects.uid = master..syslogins.suid
  and sysobjects.type = "R"
```

Standards and Compliance

Standard	Compliance Level	Comments
SQL89	Compliant	The /* style comment is a Transact-SQL extension.
SQL92	Entry level compliant	

Control-of-Flow Language

Function

Controls the order in which SQL statements, statement blocks, and stored procedures are executed.

Table 3-3 gives the control-of-flow keywords and their functions:

Keyword	Function
begin	Begins a statement block
end	Ends a statement block
break	Exits from **while** loop
continue	Restarts a **while** loop
declare	Declares local variables
goto label	Goes to a position in a statement block
if	Defines conditional execution
else	Defines alternate execution when **if** condition is false
print	Prints a user-defined message on the user's screen
raiserror	Prints a user-defined message on the user's screen and set a system flag to record the fact than an error condition has occurred
return	Exits unconditionally
waitfor	Sets delay for command execution
while	Repeats performance of statements while condition is true
/* *comment* ***/** or **--***comment*	Inserts a comment anywhere in a SQL statement

Table 3-3: Control-of-flow keywords

Standards and Compliance

Control-of-flow keywords are Transact-SQL extensions.

See Also

Commands	**begin...end**, **break**, **continue**, **create procedure**, **create trigger**, **declare**, **goto** label, **if...else**, **print**, **raiserror**, **return**, **waitfor**, **while**
Topics	Batch Queries, Comments, Parameters, Variables (Local and Global)

Cursors

Function

A cursor provides access to the set of rows returned by a SQL query. A cursor is a symbolic name that is associated with a **select** statement. Cursors enable you to access individual rows of data returned by SQL Server. Cursors consist of two parts: the **cursor result set** and the **cursor position**.

The cursor result set is the set of rows returned by the **select** statement defined for a cursor. The cursor position is the current pointer to one row within the result set.

Creating a Cursor

To create a cursor, use the **declare cursor** statement:

```
declare cursor_name cursor
for select_statement
[for {read only | update [of column_name_list]}]
```

The *cursor_name* is the name of the cursor. It must be a valid SQL Server identifier (not longer than 30 characters and must start with a letter or the symbols # or _).

select_statement is the query that defines the cursor result set. See **select** for information about its options.

The **for read only** option specifies that the cursor result set cannot be updated. In contrast, the **for update** option specifies that the cursor result set is updatable. You can specify **of** *column_name_list* after **for update** with the list of columns from the *select_statement* defined as updatable.

The following example defines a result set for the *authors_crsr* cursor that contains all authors from the *authors* table who do not reside in California:

```
declare authors_crsr cursor
for select au_id, au_lname, au_fname
from authors
where state != 'CA'
```

The **declare cursor** statement must precede any **open** statement for that cursor. You cannot include other statements with **declare cursor** in the same Transact-SQL batch.

Opening a Cursor

To open the cursor:

```
open cursor_name
```

Fetching a Row

After opening the cursor, you can fetch a row:

```
fetch cursor_name [into fetch_target_list]
```

SQL Server moves the cursor position one or more rows down the cursor result set. It retrieves the data from each row of the result set and stores the current position, allowing further fetches until it reaches the end of the result set.

into *fetch_target_list* specifies that the column data returned should be placed into the specified variables. The *fetch_target_list* must consist of Transact-SQL parameters or local variables that have been previously declared.

For example, after declaring the *@id*, *@lname*, and *@fname* variables, you can fetch rows from the *authors_crsr* as follows:

```
fetch authors_crsr into @id, @lname, @fname
```

By default, the **fetch** command only brings back one row at a time. You can use the **cursor rows** option of the **set** command to change the number of rows **fetch** returns. However, this option does not affect a **fetch** containing an **into** clause.

Closing a Cursor

When you are finished with the result set of a cursor, you can **close** it:

```
close cursor_name
```

Closing the cursor does not change its definition. You can **open** the cursor again, and SQL Server creates a new cursor result set using the same query as before.

Deallocating a Cursor

If you want to discard the cursor, you must **deallocate** it:

```
deallocate cursor cursor_name
```

Deallocating a cursor frees up any resources associated with the cursor, including the cursor name. You cannot reuse a cursor name until you deallocate it. If you deallocate an open cursor, SQL Server automatically closes it. Terminating a client connection to a server also closes and deallocates any open cursors.

Example: Using a Cursor

The following stored procedure checks the *sales* table to see if any books by a particular author have sold well. It uses a cursor to examine each row, and then prints out information. Without the cursor, it would need several **select** statements to accomplish the same task.

```
create procedure au_sales (@author_id id)
as

/* declare local variables used for fetch */
declare @title_id tid
declare @title varchar(80)
declare @ytd_sales int
declare @msg varchar(120)

/* declare the cursor to get each book written
    by given author */
declare author_sales cursor for
select ta.title_id, t.title, t.total_sales
from titleauthor ta, titles t
where ta.title_id = t.title_id
and ta.au_id = @author_id

open author_sales

fetch author_sales
        into @title_id, @title, @ytd_sales

if (@@sqlstatus = 2)
begin
    print "We do not sell books by this author."
    close author_sales
    return
end

/* if cursor result set is not empty, then process
    each row of information */
while (@@sqlstatus = 0)
begin
    if (@ytd_sales = NULL)
    begin
        select @msg = @title +
            " had no sales this year."
        print @msg
    end
    else if (@ytd_sales < 500)
```

```
        begin
            select @msg = @title +
                " had poor sales this year."
            print @msg
        end
        else if (@ytd_sales < 1000)
        begin
            select @msg = @title +
                " had mediocre sales this year."
            print @msg
        end
        else
        begin
            select @msg = @title +
                " had good sales this year."
            print @msg
        end

        fetch author_sales into @title_id, @title,
        @ytd_sales
    end

    /* if error occurred, call a designated handler */
    if (@@sqlstatus = 1) exec error_handle

    close author_sales

    deallocate cursor author_sales

    return
```

Types of Cursors

There are four kinds of cursors: **Client cursors**, **Execute cursors**, **Server cursors**, and **Language cursors**.

- Client cursors are declared through Open Client calls (or Embedded SQL). The Open Client keeps track of the rows returned from SQL Server and buffers them for the application. Updates and deletes to the result set of Client cursors can only be done through the Open Client calls.
- Execute cursors are a subset of Client cursors whose result set is defined by a stored procedure. The stored procedure can use parameters. The values of the parameters are sent through Open Client calls.
- Server cursors are declared inside a stored procedure. The client executing the stored procedure is not aware of the presence of

these cursors. Results returned to the client for a **fetch** appear exactly the same as the results from a normal **select**.

- Language cursors are declared in SQL without using Open Client. As with Server cursors, Open Client is completely unaware of the cursors and the results are sent back to the client in the same format as a normal **select**.

Client cursors, through the use of applications using Open Client calls or Embedded-SQL, are the most frequently used form of cursors. However, to simplify the discussion of cursors, the examples in this manual show Language and Server cursors. For examples of using Client or Execute cursors, see your Open Client or Embedded-SQL documentation.

Determining a Cursor's Scope

A cursor is defined by its **scope**. The scope determines the region in which the cursor is known. Once a cursor's scope no longer exists, its cursor name also no longer exists. SQL Server divides the scope of a cursor into the following regions:

- Session–this region starts when a client logs onto SQL Server and ends when it logs off. This region does not include any regions defined by stored procedures or triggers.
- Stored procedure–this region starts when a stored procedure begins execution and ends when it completes execution. If a stored procedure calls another stored procedure, SQL Server starts a new region and treats it as a subregion of the first procedure.
- Trigger–this region starts when a trigger begins execution and ends when it completes execution.

A cursor name must be unique within a given scope. Scopes are distinct in that a cursor name defined in one region can also be defined in another or within its own subregion. However, you cannot access a cursor defined in one region from another region. SQL Server does allow access to a cursor in subregions if no other cursor with that same name exists in the subregion.

Name conflicts within a particular scope are only detected by SQL Server during runtime.

For example:

```
create procedure proc1 as
    declare @flag int
    if (@flag)
        declare names_crsr cursor
        for select au_fname from authors
    else
        declare names_crsr cursor
        for select au_lname from authors
```

This stored procedure works because only one *names_crsr* cursor is defined in its scope.

Cursor Result Set and Cursor Scans

Cursor result set rows may not reflect the values in the actual base table rows. For example, a cursor declared with an **order by** clause usually requires the creation of an internal table to order the rows for the cursor result set. SQL Server does not lock the rows in the base table that correspond to the rows in the internal table, which permits other clients to update these base table rows. In that case, the rows returned to the client from the cursor result set would not be in sync with the base table rows.

A cursor result set is generated as the rows are returned through a **fetch** of that cursor. This means that a cursor **select** query is processed like a normal **select** query. This process, known as a **cursor scan**, provides a faster turnaround time and eliminates the need to read rows the application does not require.

SQL Server requires that cursor scans use a unique index of a table. However, you can still use cursors which reference tables without unique indexes, if none of those tables are updated by another process in the same page lock as that cursor. For example:

```
declare storinfo_crsr cursor
for select stor_id, stor_name, payterms
    from stores
    where state = 'CA'
```

The table *stores* specified with the above cursor does not have any indexes. SQL Server allows the declaration of cursors on tables without unique indexes, but any attempt to update these tables in the same page lock will close all cursors on such tables.

Updating or Deleting Rows Using Cursors

You can update and delete rows with cursors using the **update** or **delete** statements, but you cannot insert new rows. You can update or

delete rows for an updatable cursor. If the cursor is read-only, you can only read the data; you cannot update or delete it. By default, SQL Server attempts to determine if a cursor is updatable before designating it as read-only.

Determining Whether a Cursor Is Updatable

You can explicitly specify whether a cursor is updatable using the **read only** or **update** keywords in the **declare** statement. For example, the following defines an updatable result set for the *pubs_crsr* cursor:

```
declare pubs_crsr cursor
for select pub_name, city, state
from publishers
for update of city, state
```

This example includes all the rows from the *publishers* table, but it explicitly defines only the *city* and *state* columns for update.

If you do not explicitly specify **for read only** or **for update**, the cursor is implicitly updatable when the **select** statement does not contain any of the following:

- **distinct** option
- **group by** clause
- Aggregate functions
- **union** operator

You cannot specify the **for update** option if a cursor's *select_statement* contains one of the above constructs. SQL Server also defines a cursor as read-only if you declare a Language or Server type cursor that includes an **order by** clause as part of its *select_statement*. SQL Server handles updates differently for Client or Executable type cursors, thereby eliminating this restriction.

Determining Which Columns Can Be Updated

If you do not specify a *column_name_list* with the **for update** option, all the specified columns in the query are updatable. SQL Server attempts to use unique indexes for updatable cursors when scanning the base table. For cursors, SQL Server considers an index containing an IDENTITY column to be unique, even if it is not declared so.

If you do not specify the **for update** option, SQL Server uses any unique index, although it can also use other indexes or table scans if no unique index exists for the specified table columns. However, when you specify the **for update** option, then SQL Server must use a unique

index defined for one or more of the columns to scan the base table. If none exists, it returns an error.

For updatable cursors, SQL Server allows you to update columns that are not specified in the list of columns of the cursor's *select_statement*, but that are part of the tables specified in the *select_statement*. However, when you specify a *column_name_list* with **for update**, you can only update those specific columns.

Any columns of the base table you specify in the *column_name_list* of **for update** should include only those columns you need to update, and not any columns included in at least one unique index. This allows SQL Server to use that unique index for its cursor scan which helps prevent an update anomaly called the **Halloween Problem**.

This problem occurs when a client updates a column of a cursor result set row which defines the order in which the rows are returned from the base tables. For example, if SQL Server accesses a base table using an index and the index key is updated by the client, the updated index row can move within the index and be read again by the cursor. This is a result of an updatable cursor only logically creating a cursor result set. The cursor result set is actually the base tables that derive the cursor.

For information about how to update or delete cursor rows, see the **update** or **delete** statements in this manual.

Getting Information About Cursors

Two global variables provide information about cursor actions: *@@sqlstatus* and *@@rowcount*.

- *@@sqlstatus* holds status information (warning exceptions) resulting from the execution of a **fetch** statement. The value of *@@sqlstatus* is 0, 1, or 2.

Status	Meaning
0	Successful completion of the **fetch** statement.
1	The **fetch** statement resulted in an error.
2	There is no more data in the result set. This warning can occur if the current cursor position is on the last row in the result set and the client submits a **fetch** statement for that cursor.

Table 3-4: @@sqlstatus values

Only a **fetch** statement can set *@@sqlstatus*. All other statements have no effect on *@@sqlstatus*.

- *@@rowcount* is the number of rows returned from the cursor result set to the client up to the last **fetch**. In other words, it represents the total number of rows seen by the client at any one point in time.

  Once all the rows are read from a cursor result set, *@@rowcount* represents the total number of rows in that result set. Each open cursor is associated with a specific *@@rowcount* variable. The variable is dropped when you close the cursor. Checking *@@rowcount* after a **fetch** provides you with the number of rows read for the cursor specified in that **fetch**.

In addition to these to variables, SQL Server provides the system procedure **sp_cursorinfo**, which displays information about the cursor name, its current status (such as open or closed), and its result columns. For example:

```
sp_cursorinfo 0, authors_crsr

Cursor name 'authors_crsr' is declared at nesting
    level '0'.
The cursor is not open.
The cursor will remain open when a transaction is
    commited or rolled back.
The number of rows returned for each FETCH is 1.
The cursor is updatable.
There are 3 columns returned by this cursor.
The result columns are:
Name = 'au_id', Table = 'authors', Type = ID,
    Length = 11 (updatable)
Name = 'au_lname', Table = 'authors', Type =
    VARCHAR, Length = 40 (updatable)
Name = 'au_fname', Table = 'authors', Type =
    VARCHAR, Length = 20 (updatable)
```

For more information about **sp_cursorinfo**, see Volume 2.

Cursors and Locking

Cursor locking methods are similar to the current locking methods for SQL Server. In general, statements that read data (such as **select** or **readtext**) use shared locks on each data page to avoid reading changed data from an uncommitted transaction. Update statements use exclusive locks on each page they change. To reduce deadlocks and improve concurrency, SQL Server often precedes an exclusive lock with an update lock, which indicates that the client intends to change data on the page.

For updatable cursors, SQL Server uses update locks by default when scanning tables or views referenced with the **for update** option of

declare cursor. If the **for update** list is empty, all tables and views referenced in the **from** clause of the *select_statement* receive update locks by default.

You can instruct SQL Server to use shared locks instead of update locks by adding the **shared** keyword to that **from** clause. Specifically, you should add **shared** after each table name for which you prefer a shared lock.

➤ ***Note***

SQL Server releases an update lock when the cursor position moves off the data page. Since an application buffers rows for Client cursors, the corresponding Server cursor may be positioned on a different data row and page than the Client cursor. In this case, a second client could update the row that represents the current cursor position of the first client even if the first client used the **for update** option.

Any exclusive locks acquired by a cursor in a transaction are held until the end of that transaction. This also applies to shared or update locks when using the **holdlock** keyword or the **set isolation level 3** option. However, if you do not set the **close on endtran** option, the cursor remains open past the end of the transaction, and its current page lock remains in effect. It could also continue to acquire locks as it fetches additional rows.

For more information about locking in SQL Server, see the *System Administration Guide*.

Cursor Locking Options

These are the effects of specifying the **holdlock** or **shared** options (of the **select** statement) when defining an updatable cursor:

- If you omit both options, you can read data on the currently fetched pages only. Other users cannot update your currently fetched pages, through a cursor or otherwise. Other users can declare a cursor on the same tables you use for your cursor, but they cannot get an update lock on your currently fetched pages.
- If you specify the **shared** option, you can read data on the currently fetched pages only. Other users cannot update your currently fetched pages, through a cursor or otherwise.
- If you specify the **holdlock** option, you can read data on all pages fetched (in a current transaction) or only the pages currently fetched (if not in a transaction). Other users cannot update your

currently fetched pages or pages fetched in your current transaction, through a cursor or otherwise. Other users can declare a cursor on the same tables you use for your cursor, but they cannot get an update lock on your currently fetched pages or pages fetched in your current transaction.

- If you specify both options, you can read data on all pages fetched (in a current transaction) or only the pages currently fetched (if not in a transaction). Other users cannot update your currently fetched pages, through a cursor or otherwise.

Cursor Isolation Levels

SQL Server provides two isolation levels for cursors:

- Level 1 – SQL Server uses a shared or update lock on base table pages that contain a row representing a current cursor position. The page remains locked until the current cursor position moves off the page (as a result of **fetch** statements). If an index is used to search the base table rows, it also applies shared or update locks to those corresponding index pages. This is the default locking behavior supported by SQL Server.
- Level 3 – SQL Server uses a shared or update lock on any base table pages which have been read in a transaction on behalf of the cursor. In addition, the locks are held until the transaction ends, as opposed to releasing the locks when the data page is no longer needed. The **holdlock** keyword applies this locking level to the specified table or view.

For more information about using cursors with transactions or about isolation levels, see "Transactions" in this chapter.

See Also

Commands	**close, deallocate, declare cursor, delete, fetch, open, set, update**
System procedures	**sp_cursorinfo**

Datatypes

Function

Specify the type of information, size, and storage format of columns, stored procedure parameters, and local variables.

System-Supplied Datatypes

SQL Server provides a number of system datatypes, and allows you to build user-defined datatypes based on these types. SQL Server also supplies two user-defined datatypes, *timestamp* and *sysname*.

The table below lists the system-supplied datatypes provided for various types of information, as well as the synonyms recognized by SQL Server. For simplicity, system datatypes are printed in lowercase characters, although SQL Server allows you to enter them in either upper- or lowercase. (*timestamp* and *sysname*, like all user-defined datatypes, are case sensitive.) Most SQL Server-supplied datatypes are not reserved words, and can be used to name other objects.

Datatypes by Category	Synonyms	Range	Bytes of Storage
Exact numeric: integers			
tinyint		0 to 255	1
smallint		2^{15} -1 (32,767) to -2^{15} (-32,768)	2
int	*integer*	2^{31} (2,147,483,647) to -2^{31} (-2,147,483,648)	4
Exact numeric: decimals			
numeric (p, s)		10^{38} -1 to -10^{38}	2 to 17
decimal (p, s)	*dec*	10^{38} -1 to -10^{38}	2 to 17
Approximate numeric			
float (precision)		machine dependent	4 or 8
double precision		machine dependent	8
real		machine dependent	4

Table 3-5: Range and storage size for SQL Server system datatypes

Datatypes by Category	Synonyms	Range	Bytes of Storage
Money			
smallmoney		214,748.3647 to -214,748.3648	4
money		922,337,203,685,477.5807 to -922,337,203,685,477.5808	8
Date/time			
smalldatetime		January 1, 1900 to June 6, 2079	4
datetime		January 1, 1753 to December 31, 9999	8
Character			
char(n)	*character*	255 characters or less	*n*
varchar(n)	*character varying, char varying*	255 characters or less	actual entry length
nchar(n)	*national character, national char*	255 characters or less	*n * @@ncharsize*
nvarchar(n)	*nchar varying, national char varying, national character varying*	255 characters or less	*n*
text		2^{31} -1 (2,147,483,647) bytes or less	0 or multiple of 2K (4K on Stratus)
Binary			
binary(n)		255 bytes or less	*n*
varbinary(n)		255 bytes or less	actual entry length
image		2^{31} -1 (2,147,483,647) bytes or less	0 or multiple of 2K (4K on Stratus)
Bit			
bit		0 or 1	1 (one byte holds up to 8 *bit* columns)

Table 3-5: Range and storage size for SQL Server system datatypes (continued)

Exact Numeric Datatypes: Integers

SQL Server provides three datatypes, *tinyint*, *smallint*, and *int*, to store integers (whole numbers). These types are exact numeric types; they preserve their accuracy during arithmetic operations.

Choose among the integer types based on the expected size of the numbers to be stored. Internal storage size varies by type:

Datatype	Stores	Bytes of Storage
tinyint	Whole numbers between 0 and 255, inclusive. (Negative numbers are not permitted.)	1
smallint	Whole numbers between 2^{15} -1 and -2^{15} (32,767 and -32,768), inclusive.	2
int	Whole numbers between 2^{31} - 1 and -2^{31} (2,147,483,647 and -2,147,483,648), inclusive.	4

Table 3-6: Integer datatypes

Exact Numeric Datatypes: Decimal Numbers

SQL Server provides two other exact numeric types, *numeric* and *decimal*, for numbers that include decimal points. Data stored in *numeric* and *decimal* columns is packed to conserve disk space, and preserves its accuracy to the least significant digit after arithmetic operations. The *numeric* and *decimal* types are identical in all respects but one: only *numeric* types with a scale of 0 can be used for the IDENTITY column.

Decimal Precision and Scale

The exact numeric types accept two optional parameters, *precision* and *scale*, enclosed within parentheses and separated by a comma:

```
datatype [(precision [, scale])]
```

SQL Server treats each combination of precision and scale as a distinct datatype. For example, *numeric* (10,0) and *numeric* (5,0) are two separate datatypes. The precision and scale determine the range of values that can be stored in a *decimal* or *numeric* column:

- The precision specifies the maximum number of decimal digits that can be stored in the column. It includes all digits, to the right and left of the decimal point. You can specify a precision 1 to 38 digits, or use the default precision of 18 digits.

- The scale specifies the maximum number of digits that can be stored to the right of the decimal point. Note that the scale must be less than or equal to the precision. You can specify a scale of 0 to 38 digits, or use the default scale of 0 digits.

Exact numeric types with a scale of 0 are displayed without a decimal point. If you enter a value that exceeds either the precision or scale for the column, SQL Server returns an error message.

Storage Size

The storage size for a *numeric* or *decimal* column depends on its precision. The minimum storage requirement is 2 bytes for a 1- or 2-digit column. Storage size increases by 1 byte for each additional 2 digits of precision, up to a maximum of 17 bytes.

Approximate Numeric Datatypes

SQL Server provides three approximate numeric types, *float*, *double precision*, and *real*, for numeric data that can tolerate rounding during arithmetic operations. The approximate numeric types are especially suited to data that covers a wide range of values. They support all aggregate functions and all arithmetic operations except modulo.

Range, Precision, and Storage Size

The *real* and *double precision* types are built on types supplied by the operating system. The *float* type accepts an optional binary precision in parentheses. *float* columns with a precision of 1–15 are stored as *real*; those with higher precision are stored as *double precision*. The range and storage precision for all three types is machine dependent.

The following table shows the range and storage size for each approximate numeric type. Note that **isql** displays only six significant digits after the decimal point and rounds the remainder:

Datatype	Bytes of Storage
float[*(default precision)*]	4 for *default precision* < 16 8 for *default precision* >= 16
double precision	8
real	4

Table 3-7: Approximate numeric datatypes

Entering Approximate Numeric Data

You enter approximate numeric data as a mantissa followed by an optional exponent:

- The mantissa is a signed or unsigned number, with or without a decimal point. The column's binary precision determines the maximum number of binary digits allowed in the mantissa.
- The exponent, which begins with the character "e" or "E", must be a whole number.

The value represented by the entry is the following product:

$$\texttt{mantissa * 10}^{\texttt{exponent}}$$

For example, 2.4E3 represents the value 2.4 times 10^3, or 2,400.

Character Datatypes

The character types store strings consisting of letters, numbers, and symbols. Use the fixed-length type, *char(n)*, and the variable-length type, *varchar(n)*, for single-byte character sets such as English. Use the fixed-length type, *nchar(n)*, and the variable-length type, *nvarchar(n)*, for multibyte character sets such as Japanese. Use the *text* type (described in "text and image Datatypes") for strings longer than 255 characters. Character literals are treated as variable-length types.

You can use the **like** keyword to search character strings for particular characters, and the built-in string functions to manipulate their contents. Strings consisting of numbers can be converted to exact and approximate numeric types with the **convert** function, then used for arithmetic.

Length and Storage Size

Use *n t*o specify the length in characters for the fixed-length types, *char(n)* and *nchar(n)*. Entries shorter than the assigned length are blank-padded; entries longer than the assigned length are truncated without warning unless the **string_rtruncation** option to the **set** command is set to **on**. Fixed-length columns that allow nulls are internally converted to variable-length columns.

Use *n* to specify the maximum length in characters for the variable-length types, *varchar(n)* and *nvarchar(n)*. Data in variable-length columns is stripped of trailing blanks; storage size is the actual length of the data entered. Data in variable-length variables and parameters retains all trailing blanks, but is not padded to the full defined length.

Fixed-length columns tend to take more storage space than variable-length columns, but are accessed somewhat faster. The following table summarizes the storage requirements of the different character types:

Datatype	Stores	Bytes of Storage
char(n)	Fixed-length data, such as social security numbers or postal codes, in single-byte character sets.	*n*
nchar(n)	Variable-length data, such as names, in single-byte character sets.	*n* * *@@ncharsize*
varchar(n)	Fixed-length data in multibyte character sets	Actual number of characters entered
nvarchar(n)	Variable-length data in multibyte character sets	Actual number of characters * *@@ncharsize*
text	Up to 2,147,483,647 bytes of printable characters on linked lists of data pages	0 when uninitialized; a multiple of 2K after initialization

Table 3-8: Character datatypes

Use the **char_length** string function and **datalength** system function to determine column length. **char_length** returns the number of characters, stripping trailing blanks for variable-length types. **datalength** returns the number of bytes.

Entering Character Data

Character strings must be enclosed in single or double quotes. If you have **set quoted_identifier on**, use single quotes for character strings or SQL Server treats them as identifiers.

Strings that include the double-quote character should be surrounded by single quotes. Strings that include the single-quote character should be surrounded by double quotes. For example:

```
'George said, "There must be a better way."'
"Isn't there a better way?"
```

An alternative is to enter two quotation marks for each quotation mark you want to include in the string. For example:

```
"George said, ""There must be a better way.""
'Isn''t there a better way?'
```

To continue a character string onto the next line of your screen, enter a backslash (\) before going to the next line.

Treatment of Blank Spaces

The following example creates a table, *spaces*, with both fixed- and variable-length character columns:

```
create table spaces (cnot char(5) not null,
    cnull char(5) null,
    vnot varchar(5) not null,
    vnull varchar(5) null,
    explanation varchar(25) not null)

insert spaces values ("a", "b", "c", "d",
    "pads char-not-null only")
insert spaces values ("1    ", "2    ", "3    ",
    "4    ", "truncates trailing blanks")
insert spaces values ("    e", "    f", "    g",
    "    h", "leading blanks, no change")
insert spaces values ("   w ", "   x ", "   y ",
    "   z ", "truncates trailing blanks")
insert spaces values ("", "", "", "",
    "empty string equals space" )

select "[" + cnot + "]",
       "[" + cnull + "]",
       "[" + vnot + "]",
       "[" + vnull + "]",
    explanation from spaces
```

```
                                explanation
-------   ------- ------- ------- --------------------
[a    ]   [b]     [c]     [d]     pads char-not-null only
[1    ]   [2]     [3]     [4]     truncates trailing blanks
[    e]   [    f] [    g] [    h] leading blanks, no change
[   w ]   [   x]  [   y]  [   z]  truncates trailing blanks
[     ]   [ ]     [ ]     [ ]     empty string equals space

(5 rows affected)
```

This example illustrates how the column's datatype and null type interact to determine how blank spaces are treated:

- Only *char* **not null** and *nchar* **not null** columns are padded to the full width of the column; *char* **null** columns are treated like *varchar* and *nchar* **null** columns are treated like *nvarchar*.
- Preceding blanks are not affected.

- Trailing blanks are truncated except for *char* and *nchar* **not null** columns.
- The empty string ("") is treated as a single space. In *char* and *nchar* **not null** columns, the result is a column-length field of spaces.

Binary-Like Datatypes

The binary-like datatypes, *binary(n)*, *varbinary(n)*, and *image*, store raw binary data, such as pictures, in a hexadecimal-like notation. Binary data begins with the characters "0x" and can include any combination of digits and the uppercase and lowercase letters A through F.

➤ ***Note***

SQL Server manipulates the binary types in a platform-specific manner. For true hexadecimal data, use the *hextoint* and *inttohex* functions. For more information, see "Datatype Conversion Functions".

Use the *binary(n)* and *varbinary(n)* datatypes to store data up to 255 bytes in length. Use *n* to specify the column length in bytes, or use the default length of one byte. Each byte stores two binary digits. If you enter a value longer than *n*, SQL Server truncates the entry to the specified length without warning or error.

- Use the fixed-length binary type, *binary(n)*, for data in which all entries are expected to be approximately equal in length.
- Use the variable-length binary type, *varbinary(n)*, for data that is expected to vary greatly in length.

Because entries in *binary* columns are zero-padded to the column length (*n*), they may require more storage space than those in *varbinary* columns, but are accessed somewhat faster.

Use the *image* datatype to store larger blocks of binary data (up to 2,147,483,647 bytes) on external data pages. You cannot use the *image* datatype for variables, or for parameters in stored procedures. For more information, see the section "text and image Datatypes".

Treatment of Trailing Zeroes in binary and varbinary Data

All *binary* **not null** columns are padded with zeros to the full width of the column. Trailing zeros are truncated in all *varbinary* data, and in *binary* **null** columns since columns which accept null values must be treated as variable-length.

The following example creates a table with all four variations of *binary* and *varbinary* datatypes, NULL and NOT NULL. The same data is inserted in all four columns, and is padded or truncated according to the datatype of the column.

```
create table zeros (bnot binary(5) not null,
            bnull binary(5) null,
            vnot varbinary(5) not null,
            vnull varbinary(5) null)

insert zeros values (0x12345000, 0x12345000,
            0x12345000, 0x12345000)
insert zeros values (0x123, 0x123, 0x123, 0x123)

select * from zeros
```

```
bnot            bnull       vnot        vnull
------------    ---------   ----------  ---------
0x1234500000    0x123450    0x123450    0x123450
0x0123000000    0x0123      0x0123      0x0123
```

When you input an odd number of characters preceded by *0x*, the value is right justified and a 0 is inserted in the leftmost place. This happens because each byte of storage holds two binary digits, so all values stored must contain an even number of digits. When you enter an odd number of digits, SQL Server assumes you have omitted a leading 0.

Input values "0x00" or "0x0" are stored as "0x00" in variable-length binary columns (*binary null*, *image* and *varbinary* columns). In fixed-length binary (*binary not null*) columns, the value is padded with zeros to the full length of the field:

```
insert zeros values (0x0, 0x0,0x0, 0x0)

select * from zeros where bnot = 0x00
```

```
bnot            bnull   vnot    vnull
----------      ------  -----   ------------
0x0000000000    0x00    0x00    0x00
```

Platform Dependence

Because the exact form in which you enter a particular value depends upon the platform you are using**, calculations involving binary data can produce different results on different machines.** For platform-independent conversions between hexadecimal strings and integers, use the **inttohex** and **hextoint** functions rather than the platform-specific convert function. (See "Datatype Conversion Functions" for details.)

Money Datatypes

The money datatypes, *money* and *smallmoney*, store monetary data. You can use these types for US dollars and other decimal currencies, but SQL Server provides no means to convert from one currency to another. You can use all arithmetic operations except modulo, and all aggregate functions, with *money* and *smallmoney* data.

Both *money* and *smallmoney* are accurate to one ten thousandth of a monetary unit, but they round values up to two decimal places for display purposes. The default print format places a comma after every three digits.

The following table summarizes the range and storage requirements for money datatypes:

Datatype	Range	Bytes of Storage
money	Monetary values between +922,337,203,685,477.5807 and -922,337,203,685,477.5808	8
smallmoney	Monetary values between +214,748.3647 and -214,748.3648	4

Table 3-9: Money datatypes

Monetary values entered with E notation are interpreted as *float*. This may cause an entry to be rejected or to lose some of its precision when it is stored as a *money* or *smallmoney* value.

money and *smallmoney* values can be entered with or without a preceding currency symbol, such as a dollar sign ($). To enter a negative value, place the minus sign after the currency symbol. Do not include commas in your entry.

Date/Time Datatypes

Use the *datetime* and *smalldatetime* datatypes to store date and time information from January 1, 1753 through December 31, 9999. Dates outside this range must be entered, stored, and manipulated as *char* or *varchar* values.

- *datetime* columns hold dates between January 1, 1753 and December 31, 9999. *datetime* values are accurate to 1/300th of a second on platforms that support this level of granularity. Storage size is 8 bytes: 4 bytes for the number of days since the base date of January 1, 1900 and 4 bytes for the time of day.

- *smalldatetime* columns hold dates from January 1, 1900 to June 6, 2079, with accuracy to the minute. Storage size is 4 bytes: 2 bytes for the number of days after January 1, 1900, and 2 bytes for the number of minutes since midnight.

Display Formats

The default display format for dates is "Mon dd yyyy hh:mmAM" (or PM); for example: "Apr 15 1988 10:23PM". Use the **convert** function to display seconds and milliseconds, and to obtain additional date styles and date-part orderings by converting *datetime* data to *char*.

Entry Formats

Always enclose *datetime* and *smalldatetime* values in single or double quotes. If you have **set quoted_identifier on**, use single quotes or SQL Server treats the data as an identifier.

SQL Server recognizes a wide variety of data entry formats for date and time values. The date portion and time portion of a *datetime* values are discussed separately below. SQL Server recognizes the two portions of the data separately, so the time can precede or follow the date. Either portion can be omitted; SQL Server provides defaults, also described below. If both portions are omitted, the default date is January 1, 1900 12:00:00:000AM.

- If you specify only the last two digits of the year, values less than 50 are interpreted as "20yy", and values of 50 or greater are interpreted as "19yy".
- A date cannot begin with only a two-digit year specified. If such a date is entered, SQL Server returns an error message stating that the *datetime* value is out of range.
- You must type the century only when the day is omitted, or when you need a century other than the default, as described above.
- When entering time data, the order of the components must be as follows:

  hours
  minutes
  seconds
  milliseconds
  AM or am
  PM or pm

(12AM is midnight, 12PM is noon). To be recognized as time, a value must contain either a colon or an AM/PM signifier.

Milliseconds can be preceded either with a colon or period. If preceded by a colon, the number means thousandths of a second. If preceded by a period, a single digit means tenths of a second, two digits mean hundredths of a second, and three digits mean thousandths of a second. For example, '12:30:20:1' means twenty and one-thousandth of a second past 12:30; '12:30:20.1' means twenty and one-tenth of a second past 12:30.

Among the acceptable formats for time data are:

```
14:30
14:30[:20:999]
14:30[:20.9]
4am
4 PM
[0]4[:30:20:500]AM
```

- If you omit the time portion of a *datetime* value, the default value, 12:00:00:000 AM, is supplied.
- SQL Server recognizes three basic styles of date input. Each of the date formats shown below must be enclosed in quotes, and may be preceded or followed by a time specification, as described above.

Entering the Month in Alphabetic Format

The month can be the full month name or the month abbreviations as given in the specification for the current language. Commas are optional. Case is ignored. If the day is missing, it defaults to the first day of the month. The **dateformat** (see the **set** command) setting is not applied when you specify the month in alphabetic form.

Valid formats for specifying the date alphabetically are:

```
Apr[il] [15][,] 1988
Apr[il] 15[,] [19]88
Apr[il] 1988 [15]

[15] Apr[il][,] 1988
15 Apr[il][,] [19]88
15 [19]88 apr[il]
[15] 1988 apr[il]

1988 APR[IL] [15]
1988 [15] APR[IL]
```

Entering the Month in Numeric Format

You can enter the month in a string with slash (/), hyphen (-) or period (.) separators. The month, day, and year must be specified. The strings must be in the form:

```
<num> <sep> <num> <sep> <num> [<time spec>]
```

or:

```
[<time spec>]  <num> <sep> <num> <sep> <num>
```

The default order for date part entry is *mdy*, when the language is set to us_english. Use the **set dateformat** command to change the order for dates. The **set language** command may also affect the expected order of date parts, depending on the definition for a particular language.

The interpretation of the values of the date parts depends on the **dateformat** setting; if the ordering does not match the setting, either the values will not be interpreted as dates (because values are out of range) or the values will be misinterpreted. For example, "12/10/08" could be interpreted as one of six different dates, depending on the **dateformat** setting. See the **set** command for more information.

To enter "April 15, 1988" in *mdy* **dateformat**, you can use these formats:

```
[0]4/15/[19]88
[0]4-15-[19]88
[0]4.15.[19]88
```

The other entry orders are shown below with "/" as separators; hyphens or periods can also be used:

```
15/[0]4/[19]88  (dmy)
1988/[0]4/15  (ydm)
1988/15/[0]4  (ymd)
[0]4/[19]88/15  (myd)
15/[19]88/[0]4  (dym)
```

Entering the Date as an Unseparated String

You can enter the date as a 4, 6 or 8 digit string, or an empty string, or enter the time value, but no date value. The **dateformat** is not applied to all-numeric date entries. 6 or 8 digit strings are always interpreted as year-month-day. The month and day must always be two digits. This format is recognized:

```
[19]880415
```

If 4 digits are given, the string is interpreted as the year, and the month is set to January, the day to the first of the month. The century cannot be omitted.

An empty string ("") or missing date is interpreted as the base date, January 1, 1900. A time value without a date, for example, "4:33" is interpreted as "January 1, 1900 4:33AM".

You can do some arithmetic calculations on *datetime* values with the built-in date functions. (See "Date Functions.")

Pattern Matching with the like *Keyword*

If you use the equality operator (=) to search *datetime* values for a particular month, day and year, SQL Server returns only those values for which the time is precisely 12:00:00:000AM. Use the **like** keyword to look for dates that match a particular pattern.

For example, if you insert the value '9:20' into a column named *arrival_time*, SQL Server converts the entry into 'Jan 1 1900 9:20AM.' If you look for this entry using the equality operator, it isn't found:

```
where arrival_time = "9:20" /* does not match */
```

You can find the entry using the **like** operator:

```
where arrival_time like "%9:20%"
```

When using **like**, SQL Server first converts the dates to *datetime* format and then to *varchar*. The display format consists of the 3-character month in the current language, 2 characters for the day, and 4 characters for the year, the time in hours and minutes, and AM or PM.

You cannot use the wide variety of input formats when searching with **like**. Since the standard display formats don't include seconds or milliseconds, you cannot search for seconds or milliseconds with **like** and a match pattern, unless you are also using *style* 9 or 109 and the **convert** function.

If you are using **like**, and the day of the month is 1 through 9, you must insert two spaces between the month and day to match the *varchar* conversion of the datetime value. Similarly, if the hour is less than 10, the conversion places two spaces between the year and the hour. The clause:

```
like May 2%
```

(with one space between "May" and "2") finds all dates from May 20 through May 29, but not May 2. You do not need to insert the extra space with other date comparisons, only with **like**, since the datetime values are converted to *varchar* only for the **like** comparison.

The *bit* Datatype

Use *bit* columns for true / false or yes / no types of data. *bit* columns hold either 0 or 1. Integer values other than 0 or 1 are accepted, but are always interpreted as 1. Storage size is 1 byte. Multiple *bit* datatypes in a table are collected into bytes. For example, 7 *bit* columns fit into 1 byte; 9 *bit* columns take 2 bytes.

Columns of datatype *bit* cannot be NULL and cannot have indexes on them. The *status* column in the *syscolumns* system table indicates the unique offset position for bit columns.

The *timestamp* User-Defined Datatype

timestamp is a SYBASE-supplied user-defined datatype, defined as *varbinary(8)* NULL. *timestamp* columns are necessary in tables that are to be browsed (see "Browse Mode") in Client Library applications.

Because *timestamp* is a user-defined datatype, you cannot use it to define other user-defined datatypes. You must enter it as shown here, in all lowercase letters.

Every *time* a row containing a *timestamp* column is inserted or updated, SQL Server automatically updates the *timestamp* column. A table can have only one column of *timestamp* datatype.

Creating a timestamp Column

If you create a column named *timestamp* without specifying a datatype, SQL Server automatically defines the column as a *timestamp* datatype:

```
create table testing
    (c1 int, timestamp, c2 int)
```

You can also explicitly assign the *timestamp* datatype to a column named *timestamp*:

```
create table testing
    (c1 int, timestamp timestamp, c2 int)
```

or to a column with another name:

```
create table testing
    (c1 int, t_stamp timestamp,c2 int)
```

You can create a column named *timestamp* and assign it another datatype (although this could be confusing to other users, and would

not allow the use of the **browse** functions in Open Client or with the **tsequal** function):

```
create table testing
    (c1 int, timestamp datetime)
```

The *sysname* User-Defined Datatype

sysname is a user-defined datatype that is distributed on the SQL Server installation tape and used in the system tables. Its definition is:

```
varchar(30) "not null"
```

You cannot declare a column, parameter, variable to be of type *sysname*. It is possible, however, to create a user-defined datatype with a base type of *sysname*. You can then define columns, parameters, and variables with the user-defined type.

User-Defined Datatypes

SQL Server allows you to create user-defined datatypes, based on any system datatype, with the **sp_addtype** system procedure. You cannot create a user-defined datatype based on another user-defined datatype, such as *timestamp* or the *tid* datatype in the *pubs2* database. *sysname*, as described above, is a special case.

User-defined datatypes are database objects. Their names are case sensitive and must conform to the rules for identifiers. Use **sp_rename** to rename a user-defined datatype, and **sp_droptype** to remove a user-defined datatype from a database.

➤ ***Note***

You cannot drop a datatype that is already in use in any table.

In addition to datatype information (including length, precision, and scale where appropriate), user-defined datatypes can specify a default null type or the IDENTITY property. You can bind rules to user-defined datatypes with **sp_bindrule**, and bind defaults with **sp_bindefaul**t.

After creating a user-defined datatype in a database, you can use it to create columns, parameters, and variables.These automatically inherit the datatype, rules, and defaults associated with the user-defined type. By default, they inherit the null type or IDENTITY property of the user-defined type, although you can override these in a column definition.

A user-defined datatype must be created in each database in which it will be used. It is a good practice to create frequently used types in the *model* database. These types are automatically be added to each new database (including *tempdb*, which is used for temporary tables) as it is created.

Getting Help on Datatypes

Use the **sp_help** system procedure to display information about the properties of a system or user-defined datatype. You can also use this procedure to display the datatype, length, precision, and scale for each column in a table.

Declaring the Datatype of a Column, Variable, or Parameter

You must declare the datatype for a column, local variable, or parameter. The datatype can be any of the system-supplied datatypes or any user-defined datatype in the database.

Declaring the Datatype for a Column in a Table

Use the following syntax to declare the datatype of a new column in an **alter table** or **create table** statement:

```
create table [[database.]owner.]table_name
   (column_name datatype [identity | not null | null]
       [, column_name datatype [identity | not null |
           null]]...)

alter table [[database.]owner.]table_name
   add column_name datatype [identity | null
       [, column_name datatype [identity | null]...
```

For example:

```
create table sales_daily
    (stor_id char(4)not null,
     ord_num numeric(10,0)identity,
     ord_amt money null)
```

Declaring the Type for a Local Variable in a Batch or Procedure

Use the following syntax to declare the datatype for a local variable in a batch or stored procedure:

```
declare @variable_name datatype
   [, @variable_name datatype]...
```

For example:

```
declare @hope money
```

Declaring the Type for a Parameter in a Stored Procedure

Use the following syntax to declare the datatype for a parameter in a stored procedure:

```
create procedure [owner.]procedure_name [;number]
   [[(]@parameter_name datatype [= default] [output]
       [,@parameter_name datatype [= default]
           [output]]...[)]]
   [with recompile]
   as SQL_statements
```

For example:

```
create procedure auname_sp @auname varchar(40)
as
    select au_lname, title, au_ord
    from authors, titles, titleauthor
    where @auname = au_lname
    and authors.au_id = titleauthor.au_id
    and titles.title_id = titleauthor.title_id
```

Determining the Datatype of a Literal

You cannot declare the datatype of a literal. SQL Server treats all character literals as *varchar*. Numeric literals entered with E notation are treated as *float*; all others as exact numerics:

- Literals between $2^{31} - 1$ and -2^{31} with no decimal point are treated as *integer*
- Literals that include a decimal point, or that fall outside the range for integers, are treated as *numeric*

➤ ***Note***

To preserve backward compatibility, use E notation for numeric literals that should be treated as floats.

Datatype of Mixed-Mode Expressions

When you perform concatenation or mixed-mode arithmetic on values with different datatypes, SQL Server must determine the datatype and, in some cases, the length and precision, of the result.

Determining Datatype Hierarchy

Each system datatype has a **datatype hierarchy**, which is stored in the *systypes* system table. User-defined datatypes inherit the hierarchy of the system type on which they are based.

The following query ranks the datatypes in a database by hierarchy. In addition to the information shown below, your query results will include information about any user-defined datatypes in the database:

```
select name,hierarchy
from systypes
order by hierarchy
```

```
name                          hierarchy
-----------------------------  ---------
floatn                               1
float                                2
datetimn                             3
datetime                             4
real                                 5
numericn                             6
numeric                              7
decimaln                             8
decimal                              9
moneyn                              10
money                               11
smallmoney                          12
smalldatetime                       13
intn                                14
int                                 15
smallint                            16
tinyint                             17
bit                                 18
varchar                             19
sysname                             19
nvarchar                            19
char                                20
nchar                               20
varbinary                           21
timestamp                           21
binary                              22
text                                23
image                               24
(28 rows affected)
```

The datatype hierarchy determines the results of computations using values of different datatypes. The result value is assigned the datatype that is closest to the top of the list.

In the following example, *qty* from the *sales* table is multiplied by *royalty* from the *roysched* table. *qty* is a *smallint*, which has a hierarchy of 16; *royalty* is an *int*, which has a hierarchy of 15. Therefore, the datatype of the result is an *int*.

```
smallint(qty) * int(royalty) = int
```

Determining Precision and Scale

For the *numeric* and *decimal* types, each combination of precision and scale is a distinct SQL Server datatype. If you perform arithmetic on two *numeric* or *decimal* values

- *n1* with precision *p1* and scale *s1*
- and *n2* with precision *p2* and scale *s2*

SQL Server determines the precision and scale of the results as follows:

Operation	Precision	Scale
n1 + n2	max(s1, s2) + max(p1 -s1, p2 - s2) + 1	max(s1, s2)
n1 - n2	max(s1, s2) + max(p1 -s1, p2 - s2) + 1	max(s1, s2)
n1 * n2	s1 + s2 + (p1 - s1) + (p2 - s2) + 1	s1 + s2
n1 / n2	max(s1 + p2 -s2 + 1, 6) + p1 - s1 + p2	max(s1 + p2 -s2 + 1, 6)

Table 3-10: Precision and scale after arithmetic operations

Converting One Datatype to Another

Many conversions from one datatype to another are handled automatically by SQL Server. These are called implicit conversions. Other conversions must be performed explicitly with the **convert**, **inttohex**, and **hextoint** functions. See "Datatype Conversion Functions" in Chapter 2 for details about datatype conversions supported by SQL Server.

Automatic Conversion for Fixed-Length NULL Columns

Only columns with variable-length datatypes can store null values. When you create a NULL column with a fixed-length datatype, SQL Server automatically converts it to the corresponding variable-

length datatype. SQL Server does not inform the user of the type change.

The following chart lists the fixed-length datatypes and the variable-length datatypes to which they are converted. Certain variable-length datatypes, such as *moneyn*, are reserved types; you cannot use them to create columns, variables, or parameters:

Original Fixed-Length Datatype	Converted To
char	*varchar*
nchar	*nvarchar*
binary	*varbinary*
datetime	*datetimn*
float	*floatn*
int, *smallint*, and *tinyint*	*intn*
decimal	*decimaln*
numeric	*numericn*
money and *smallmoney*	*moneyn*

Table 3-11: Automatic conversion of fixed-length to variable-length datatypes

Handling Overflow and Truncation Errors

The **arithabort** option determines how SQL Server behaves when an arithmetic error occurs. The two **arithabort** options, **arithabort arith_overflow** and **arithabort numeric_truncation**, handle different types of arithmetic errors. You can set each option independently, or set both options with a single **set arithabort on** or **set arithabort off** statement.

- **arithabort arith_overflow** specifies behavior following a divide-by-zero error or a loss of precision during either an explicit or an implicit datatype conversion. This type of error is considered serious. The default setting, **arithabort arith_overflow on**, rolls back the entire transaction or batch in which the error occurs. If you set **arithabort arith_overflow off**, SQL Server aborts the statement that causes the error but continues to process other statements in the transaction or batch.
- **arithabort numeric_truncation** specifies behavior following a loss of scale by an exact numeric type during an implicit datatype conversion. (When an explicit conversion results in a loss of scale, the results are truncated without warning.) The default setting, **arithabort numeric_truncation on**, aborts the statement that causes the error but continues to process other statements in the transaction or batch. If you set **arithabort numeric_truncation off**, SQL Server truncates the query results and continues processing.

The **arithignore** option determines whether SQL Server prints a warning message after an overflow error. By default, the **arithignore** option is turned **off**. This causes SQL Server to display a warning message after any query that results in numeric overflow. To ignore overflow errors, **set** the **arithignore** option **on**.

➤ ***Note***

The **arithabort** and **arithignore** options have been redefined for Release 10.0. If you use these options in your applications, examine them to be sure they still produce the desired effects.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL provides the *smallint*, *int*, *numeric*, *decimal*, *float*, *double precision*, *real*, and *char* SQL89 datatypes. The *tinyint*, *binary*, *varbinary*, *image*, *bit*, *datetime*, *smalldatetime*, *money*, *smallmoney*, *nchar*, *nvarchar*, *sysname*, *text*, *timestamp*, *varchar*, and user-defined datatypes are Transact-SQL extensions.
SQL92	Transact-SQL provides the *smallint*, *int*, *numeric*, *decimal*, *float*, *double precision*, *real*, *char*, and *varchar* SQL89 datatypes. The *tinyint*, *binary*, *varbinary*, *image*, *bit*, *datetime*, *smalldatetime*, *money*, *smallmoney*, *nchar*, *nvarchar*, *sysname*, *text*, *timestamp*, and user-defined datatypes are Transact-SQL extensions.

See Also

Commands	**create procedure**, **create table**, **execute**, **set**
Functions	Aggregate Functions, Datatype Conversion Functions, Date Functions, Mathematical Functions, String Functions, *text* and *image* Functions
Topics	Parameters, *text* and *image* Datatypes, Variables (Local and Global)
System procedures	**sp_addtype, sp_bindefault, sp_bindrule, sp_droptype, sp_help, sp_rename, sp_unbindefault, sp_unbindrule**

Disk Mirroring

Function

Creates a software mirror of a user database device, the *master* database device, or a database device used for user database transaction logs. If a database device fails, its mirror immediately takes over.

Disk mirroring does not interfere with ongoing activities in the database. You can mirror or unmirror database devices without shutting down SQL Server.

Deciding What to Mirror

You should mirror all default database devices so that you are still protected if a **create** or **alter database** command affects a database device in the default list.

In addition to mirroring user database devices, you should always put their transaction logs on a separate database device. You can also mirror the database device used for transaction logs, for even greater protection.

To put a database's transaction log (that is, the system table *syslogs*) on a different device than the one on which the rest of the database is stored, name the database device and the log device when you create the database. You could also **alter database** to a second device and then run the system procedure **sp_logdevice**.

Mirroring a Device

Use the **disk mirror** command to mirror a database device. Its syntax is:

```
disk mirror
   name = "device_name" ,
   mirror = "physicalname"
   [ ,writes = { serial | noserial }]
   [ ,contiguous ] (OpenVMS only)
```

For example, to create a software mirror for the database device *user_disk* on the file *mirror.dat*:

```
disk mirror
  name = "user_disk",
  mirror = "/server/data/mirror.dat"
```

Specifying the Mirror Device

Use the **mirror =** *"physicalname"* clause to specify the path to the mirror device, enclosed in single or double quotes. If the mirror device is a file, *physicalname* must unambiguously identify the path where SQL Server will create the file; it cannot specify the name of an existing file.

A device and its mirror together constitute one logical device. The mirror device is used only as a secondary device and does not require a separate entry in *sysdevices*. SQL Server stores the physical name of the mirror device in the *mirrorname* column of the *sysdevices* table.

Overriding Serial Writes

By default (**writes = serial**), the write to the primary database device is guaranteed to finish before the write to the secondary device begins. When the primary and secondary devices are on different physical devices, serial writes help ensure that one of the disks is unaffected by a power failure.

Use the **writes = noserial** option to override serial writes.

Requiring the Mirror File to Be Contiguous (OpenVMS Only)

When creating a file as a secondary device, SQL Server tries to use contiguous space. If not enough contiguous blocks are found, the file is created discontiguously and a message printed.

Use the **contiguous** option to force secondary files to be created contiguously. If you include the **contiguous** option, the system creates a contiguous file or the command fails with an error message.

Automatic Unmirroring After Device Failure

When a read or write to a mirrored device is unsuccessful, SQL Server causes the bad device to become unmirrored, and prints error messages. SQL Server continues to run, unmirrored. The System Administrator must remirror the disk to restart mirroring.

Disabling a Mirror

Use the **disk unmirror** command to disable a mirror, either temporarily or permanently. Its syntax is:

```
disk unmirror
   name = "device_name"
   [ ,side = { primary | secondary }]
   [ ,mode = { retain | remove }]
```

For example, to suspend software mirroring for the database device *user_disk*:

```
disk unmirror
  name = "user_disk"
```

Choosing Which Device to Disable

Use the **side** clause to specify which device of a mirrored pair to disable. By default, SQL Server disables the secondary device (mirror). Use **side=primary** to disable the device listed in the *phyname* column of *sysdevices*.

Temporarily Deactivating a Device

By default **(mode=retain),** SQL Server deactivates the specified device temporarily; you can reactivate it at a later time. This is similar to what happens when a device fails and SQL Server activates its mirror:

- I/O is directed only at the remaining device of the mirrored pair.
- The *status* column of *sysdevices* is altered to indicate that the mirroring feature has been deactivated.
- The entries for primary *(phyname)* and secondary *(mirrorname*) disks are unchanged.

Permanently Disabling a Mirror

Use **mode=remove** to completely disable disk mirroring. This option eliminates all references in the system tables to a mirror device. If you set **mode=remove**:

- The *status* column is altered to indicate that the mirroring feature is to be completely ignored.
- The *phyname* column is replaced by the name of the secondary device in the *mirrorname* column if the primary device is the one being deactivated.

- The *mirrorname* column is set to NULL.

➤ ***Note***

mode=remove does not remove an operating system file that has been used as a mirror.

Re-enabling a Mirror

Use the **disk remirror** command to re-enable a mirror that has been temporarily disabled by the **disk unmirror** command or by device failure. Its syntax is:

```
disk remirror
   name = "device_name"
```

For example, to resume software mirroring on the database device *user_disk*:

```
disk remirror
  name = "user_disk"
```

Backing Up *master* after the Disk Mirroring Commands

It is important to back up the *master* database with the **dump database** command after each use of **disk mirror, disk unmirror,** or **disk remirror.** This makes recovery easier and safer in case *master* is damaged.

Mirroring the Master Device

If you mirror the database device for the *master* database, you can use the **-r** option and the name of the mirror for UNIX, or the **/mastermirror** option for OpenVMS, when you restart SQL Server with the **dataserver** utility program. Add this option to the *RUN_servername* file for that server, so that the **startserver** utility program will know about it. For example:

```
dataserver -dmaster.dat -rmirror.dat
```

starts a master device named *master.dat* and its mirror, *mirror.dat*. For more information, see **dataserver** and **startserver** in the *SQL Server Utility Programs* for your platform.

Getting Information About Devices and Mirrors

For a report on all SQL Server devices on your system (user database devices and their mirrors, as well as dump devices), execute the system procedure **sp_helpdevice**.

See Also

Commands	**alter database, create database, disk init, disk mirror, disk refit, disk reinit, disk remirror, disk unmirror, dump database, dump transaction, load database, load transaction**
System procedures	**sp_diskdefault, sp_helpdevice, sp_logdevice**
Utility programs	**dataserver, startserver**

Expressions

Function

An expression is a combination of one or more constants, literals, functions, column identifiers and / or variables, separated by operators, that returns a single value. Expressions can be of several types, including **arithmetic**, **relational**, **logical** (or **Boolean**), and **character string**. In some Transact-SQL clauses, a subquery can be used in an expression.

Arithmetic and Character Expressions

The general pattern for arithmetic and character expressions is:

```
{constant | column_name | function | (subquery)}
    [{arithmetic_operator | bitwise_operator |
     string_operator | comparison_operator }
{constant | column_name | function | (subquery)}]...
```

Relational and Logical Expressions

A logical expression or relational expression returns TRUE, FALSE or UNKNOWN. The general patterns are:

```
expression comparison_operator [any | all] expression

expression [not] in expression

[not]exists expression

expression [not] between expression and expression

expression [not] like "match_string"
   [escape "escape_character"]

not expression like "match_string"
   [escape "escape_character"]

expression is [not] null

not logical_expression

logical_expression {and | or} logical_expression
```

Operator Precedence

Operators have the following precedence levels, where 1 is the highest level and 6 is the lowest:

1. unary (single argument) - + ~
2. * / %
3. binary (two argument) + - & | ^
4. **not**
5. **and**
6. **or**

When all operators in an expression are of the same level, the order of execution is left to right. You can change the order of execution with parentheses—the most deeply nested expression is handled first.

Arithmetic Operators

SQL Server uses the following arithmetic operators:

Symbol	Meaning
+	addition
–	subtraction
*	multiplication
/	division
%	modulo (Transact-SQL extension)

Table 3-12: Arithmetic operators

Addition, subtraction, division, and multiplication can be used on *int*, *smallint*, *tinyint*, *float*, *real*, *smallmoney* and *money* columns.

The modulo operator cannot be used on *smallmoney*, *money*, *float* or *real* columns. Modulo finds the integer remainder after a division involving two whole numbers. For example, 21 % 11 = 10 because 21 divided by 11 equals 1 with a remainder of 10.

When you perform arithmetic operations on mixed datatypes, for example *float* and *int*, the Server follows specific rules for determining the type of the result. See “Datatypes” for more information.

Bitwise Operators

The bitwise operators are a Transact-SQL extension for use with integer type data. These operators convert each integer operand into its binary representation, then evaluate the operands column by column. A value of 1 corresponds to true, a value of 0 to false.

The following tables summarize the results for operands of 0 and 1. If either operand is NULL, the bitwise operator returns NULL:

& (and)	1	0
1	1	0
0	0	0

| (or)	1	0
1	1	1
0	1	0

^ (exclusive or)	1	0
1	0	1
0	1	0

~ (not)	
1	FALSE
0	0

Table 3-13: Truth tables for bitwise operations

The following examples use two integer arguments, A = 170 (10101010 in binary form) and B = 75 (01001011 in binary form):

Operation	Binary Form	Result	Explanation
(A & B)	10101010 01001011 ---------- 00001010	10	Result column equals 1 if both A and B are 1. Otherwise, result column equals 0.
(A | B)	10101010 01001011 ---------- 11101011	235	Result column equals 1 if either A or B, or both, is 1. Otherwise, result column equals 0.
(A ^ B)	10101010 01001011 ---------- 11100001	225	Result column equals 1 if either A or B, but not both, is 1.
(~A)	10101010 ---------- 01010101	85	All 1's are changed to 0's and all 0's to 1's.

Table 3-14: Examples of bitwise operations

The String Concatenation Operator

The string operator **+** can be used to concatenate two or more character or binary expressions. For example:

1.
```
select Name = (au_lname + ", " + au_fname)
from authors
```

   Displays author names under the column heading *Name* in last-name first-name order, with a comma after the last name, for example, *Bennett, Abraham*.

2.
```
select "abc" + "" + "def"
```

   Returns the string "abc def". The empty string is interpreted as a single space in all *char*, *varchar*, *nchar*, *nvarchar*, and *text* concatenation, and in *varchar* insert and assignment statements.

When concatenating non-character, non-binary expressions, always use **convert**:

```
select "The date is " +
    convert(varchar(12), getdate())
```

The Comparison Operators

SQL Server uses the following comparison operators:

Symbol	Meaning
=	equal to
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
<>	not equal to
!=	not equal to (Transact-SQL extension)
!>	not greater than (Transact-SQL extension)
!<	not less than (Transact-SQL extension)

Table 3-15: Comparison operators

In comparing character data, **<** means closer to the beginning of the server's sort order and **>** means closer to the end of the sort order. Upper and lower case letters are equal in a case-insensitive sort order. Use **sp_helpsort** to see the sort order for your SQL Server. Trailing blanks are ignored for comparison purposes. So, for example, "Dirk" is the same as "Dirk ".

In comparing dates, **<** means earlier and **>** means later.

Put single or double quotes around all *character* and *datetime* data used with a comparison operator:

```
= "Bennet"
> "May 22 1947"
```

Nonstandard Operators

The following operators are Transact-SQL extensions:

- Modulo operator: %
- Negative comparison operators: !>, !<, !=
- Bitwise operators: ~, ^, |, &
- Join operators: *= and =*

Using any, all and in

any is used with **<, >**, or **=** and a subquery. It returns results when any value retrieved in the subquery matches the value in the **where** or **having** clause of the outer statement. See "Subqueries" for more information.

all is used with **<** or **>** and a subquery. It returns results when all values retrieved in the subquery are less than (**<**) or greater than (**>**) the value in the **where** or **having** clause of the outer statement. See "Subqueries" for more information.

in returns results when any value returned by the second expression matches the value in the first expression. The second expression must be a subquery or a list of values enclosed in parentheses. **in** is equivalent to **= any**. See "**where** Clause" for details.

Negating and Testing

not negates the meaning of a keyword or logical expression.

Use **exists** to test for the existence of some result. The following expression must be a subquery.

Ranges

between is the range-start keyword; **and** is the range-end keyword. The range:

```
where column1 between x and y
```

is inclusive. The range:

```
where column1 > x and column1 < y
```

is not inclusive.

Pattern Matching

like indicates that the following expression is a matching pattern. With **like**, the second expression must always be a character string. **like** is available for all character and *datetime* datatypes (but not to search for seconds or milliseconds).

Using Wildcards

A *match_string* is a character string that can be used with a number of wildcards:

Symbol	Meaning
%	any string of 0 or more characters
_	any single character
[ ]	any single character within the specified range ([a-f]) or set ([abcdef])
[^]	any single character not within the specified range ([^a-f]) or set ([^abcdef])

Table 3-16: Wildcards used with like

The wildcard and the match string must be enclosed in single or double quotes (**like** "[dD]eFr_nce"). For complete information, including information on using alternative character set definitions in wildcards, see "Wildcard Characters."

Escape Characters

The **escape** keyword specifies an escape character with which you can search for literal occurrences of wildcard characters. See "Wildcard Characters" for a complete explanation.

Using Nulls

Use **is null** or **is not null** in queries on columns defined to allow null values. If **null** is used to query columns defined as **not null**, SQL Server displays an error message.

An expression with a bitwise or arithmetic operator evaluates to NULL if any of the operands are null. See "Null Values" for more information.

Connecting Expressions

and connects two expressions and returns results when both are true. When more than one logical operator is used in a statement, **and** is handled first. However, you can change the order of execution with parentheses.

or connects two or more conditions and returns results when either of the conditions is true. When more than one logical operator is used in a statement, **or** is evaluated after **and**. However, you can change the order of execution with parentheses.

This table shows the results of logical operations, including those which involve null values:

and	TRUE	FALSE	NULL
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
NULL	UNKNOWN	FALSE	UNKNOWN

or	TRUE	FALSE	NULL
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN

Table 3-17: Truth tables for logical expressions

and	TRUE	FALSE	NULL
NULL	TRUE	UNKNOWN	UNKNOWN

not	
TRUE	FALSE
FALSE	TRUE
NULL	UNKNOWN

Table 3-17: Truth tables for logical expressions (continued)

The result UNKNOWN indicates that one or more of the expressions evaluates to NULL, and that the result of the operation cannot be determined to be either TRUE or FALSE. See "Null Values" for more information.

Using Parentheses in Expressions

Parentheses can be used to group the elements in an expression. When "expression" is given as a variable in a syntax statement, a simple expression is assumed. "Logical expression" is specified when only a logical expression is acceptable.

Comparing Character Expressions

Character constant expressions are treated as *varchar*. If they are compared with non-*varchar* variables or column data, the datatype precedence rules are used in the comparison, that is, the "lower" datatype is converted to the "higher" datatype. If implicit conversion of the lower datatype to the higher datatype is not allowed, you must use **convert**.

Comparison of a *char* expression to a *varchar* expression follows the datatype precedence rule; the "lower" datatype is converted to the "higher" datatype. All *varchar* expressions are converted to *char* (that is, trailing blanks are appended) for the comparison.

Using the Empty String

The empty string ("") or ('') is interpreted as a single blank in **insert** or assignment statements on *varchar* data. In concatenation of *varchar, char, nchar, nvarchar* data, the empty string is interpreted as a single space; for example:

"**abc**" + "" + "**def**"

is stored as "abc def". The empty string is never evaluated as NULL.

Including Quotation Marks in Character Expressions

There are two ways to specify literal quotes within a *char* or *varchar* entry. The first method is to double the quotes. For example, if you began a character entry with a single quote but wanted to include a single quote as part of the entry, use two single quotes:

```
'I don''t understand.'
```

With double quotes:

```
"He said, ""It's not really confusing."""
```

The second method is to enclose a quote in the opposite kind of quote mark. In other words, surround an entry containing a double quote with single quotes (or vice versa). Here are some examples:

```
'George said, "There must be a better way."'
"Isn't there a better way?"
'George asked, "Isn"t there a better way?"'
```

Using the Continuation Character

To continue a character string to the next line on your screen, enter a backslash (\ ) before going to the next line.

See Also

Commands	**select**, **where** Clause
Topics	Subqueries, Wildcard Characters
System procedures	**sp_helpjoins**

Identifiers

Function

Identifiers are names for database objects such as databases, tables, views, columns, indexes, triggers, procedures, defaults, rules, and cursors.

Valid Identifiers

SQL Server identifiers can be a maximum of 30 bytes in length, whether single-byte or multibyte characters are used. The first character of an identifier must be either an alphabetic character as defined in the current character set or the underscore (_) symbol.

➤ ***Note***

Temporary table names begin with # and local variable names with @.

Subsequent characters can include letters, numbers, the symbols #, @, _, or currency symbols such as $ (dollars), ¥ (yen) and £ (pound sterling). Identifiers cannot include special characters such as !%^&*. or embedded spaces.

You cannot use a reserved word, such as a Transact-SQL command, as an identifier. For a complete list of reserved words, see Appendix A, ‘‘Reserved Words.’’

Object Names That Begin with # (Temporary Tables)

Tables whose names begin with # are temporary tables. You cannot create other types of objects whose names begin with the pound sign.

SQL Server performs special operations on temporary table names to maintain unique naming on a per-session basis. Long temporary tables names are truncated to 13 characters (including the pound sign); short names are padded to 13 characters with underscores (_). A 17-digit numeric suffix is appended that is unique for a SQL Server session.

Case Sensitivity

SQL Server’s sensitivity to the case (upper or lower) of identifiers and data depends on the sort order installed on your Server. Case sensitivity can be changed for single-byte character sets by reconfiguring SQL Server’s sort order. (See the *System Administration*

Guide for more information). Case is significant in utility program options.

If SQL Server is installed with a case-insensitive sort order, you cannot create a table named *MYTABLE* if a table named *MyTable* or *mytable* already exists. Similarly, this command:

```
select * from MYTABLE
```

will return rows from *MYTABLE*, *MyTable*, or *mytable*, or any combination of upper and lower case letters in the name.

Uniqueness

Object names need not be unique in a database. However, column names and index names must be unique within a table, and other object names must be unique for each owner within a database. Database names must be unique on a SQL Server.

Using Delimited Identifiers

Delimited identifiers are object names enclosed in double quotes. Using delimited identifiers allows you to avoid certain restrictions on object names. Table, view, and column names can be delimited by quotes; other object names cannot.

Delimited identifiers can be reserved words, can begin with non-alphabetic characters, and can include characters that would not otherwise be allowed. They cannot exceed 28 bytes.

◆ ***WARNING!***

Delimited identifiers may not be recognized by all front-ends and should not be used as parameters to system procedures.

Before creating or referencing a delimited identifier, you must execute:

```
set quoted_identifier on
```

Each time you use the delimited identifier in a statement, you must enclose it in double quotes. For example:

```
create table "1one"(col1 char(3))
create table "include spaces" (col1 int)

create table "grant"("add" int)
insert "grant"("add") values (3)
```

While the **quoted_identifier** option is turned on, do not use double quotes around character or date strings; use single quotes instead.

Delimiting these strings with double quotes causes SQL Server to treat them as identifiers. For example, to insert a character string into *col1* of *1table*, use:

```
insert "1one"(col1) values ('abc')
```

not:

```
insert "1one"(col1) values ("abc")
```

To insert a single quote into a column, use two consecutive single quotation marks. For example, to insert the characters a'b into *col1* use:

```
insert "1one"(col1) values('a''b')
```

Using Qualified Object Names

You can uniquely identify a table or column by adding other names that qualify it—the database name, owner's name, and (for a column) the table or view name. Each of these qualifiers is separated from the next by a period. For example:

```
database.owner.table_name.column_name
```

```
database.owner.view_name.column_name
```

The naming conventions are:

```
[[database.]owner.]table_name
```

```
[[database.]owner.]view_name
```

Using Delimited Identifiers Within an Object Name

If the **quoted_identifier** option is set **on**, you can use double quotes around individual parts of a qualified object name. Use a separate pair of quotes for each qualifier that requires quotes. For example, use:

```
database.owner."table_name"."column_name"
```

rather than:

```
database.owner."table_name.column_name"
```

Omitting the Owner Name

You can omit the intermediate elements in a name and use dots to indicate their positions, as long as the system is given enough information to identify the object:

```
database..table_name
```

```
database..view_name
```

Referencing Your Own Objects in the Current Database

You need not use the database name or owner name to reference your own objects in the current database. The default value for *owner* is the current user, and the default value for *database* is the current database.

If you reference an object without qualifying it with the database name and owner name, SQL Server tries to find the object in the current database among the objects you own.

Referencing Objects Owned by the Database Owner

If you omit the owner name and you don't own an object by that name, SQL Server looks for objects of that name owned by the Database Owner. You must qualify objects owned by the Database Owner only if you own an object of the same name, but want to use the object owned by the Database Owner. However, you must qualify objects owned by other users with the user's name, whether you own objects of the same name or not.

Using Qualified Identifiers Consistently

When qualifying a column name and table name in the same statement, be sure to use the same qualifying expressions for each; they are evaluated as strings and must match or an error is returned. The second of the following examples is incorrect because the syntax style for the column name does not match the syntax style used for the table name.

```
1. select demo.mary.publishers.city
   from demo.mary.publishers

   city
   ----------------------
   Boston
   Washington
   Berkeley

2. select demo.mary.publishers.city
   from demo..publishers

   The column prefix "demo.mary.publishers" does not
   match a table name or alias name used in the query.
```

Determining Whether an Identifier Is Valid

You can use the system function **valid_name** to determine if an identifier you have created is acceptable to SQL Server. Here is the syntax:

```
select valid_name("string")
```

where *string* is the identifier you wish to check. If *string* is not valid as an identifier, SQL Server returns a 0 (zero). If *string* is a valid identifier, SQL Server returns a nonzero number. SQL Server returns a 0 if the characters used are illegal or if *string* is more than 30 bytes long. Use this function to check object names after changing character sets, or to verify names before creating tables and other database objects.

Renaming Database Objects

Rename user objects (including user-defined datatypes) with **sp_rename**.

◆ ***WARNING!***

After you rename a table or column, be sure to redefine any procedures, triggers, and views that depend on the renamed object.

Using Multibyte Character Sets

In multibyte character sets, a wider range of characters is available for use in identifiers. For example, on a server with the Japanese language installed, the following types of characters may be used as the first character of an identifier: Zenkaku or Hankaku Katakana, Hiragana, Kanji, Romaji, Greek, Cyrillic, or ASCII.

Although Hankaku Katakana characters are legal in identifiers on Japanese systems, they are not recommended for use in heterogeneous systems. These characters cannot be converted between the EUC-JIS and Shift-JIS character sets.

The same is true for some 8-bit European characters. For example, the character "Œ," the OE ligature, is part of the Macintosh character set (codepoint 0xCE). This character does not exist in the ISO 8859-1 character set. If "Œ" exists in data being converted from the Macintosh to the ISO 8859-1 character set, it causes a conversion error.

If an object identifier contains a character that cannot be converted, the client loses direct access to that object.

Standards and Compliance

Standard	Compliance Level
SQL89	To be compliant, identifiers must not: • Begin with the pound sign (#) • Have more than 18 characters • Contain lowercase letters
SQL92	To be entry level compliant, identifiers must not: • Begin with the pound sign (#) • Have more than 18 characters

See Also

Commands	**create database**, **create default**, **create procedure**, **create rule**, **create table**, **create trigger**, **create view**, **select**, **set**
Functions	System Functions
Topics	Datatypes
System procedures	**sp_rename**

IDENTITY Columns

Function

IDENTITY columns contain system-generated values that uniquely identify each row within a table. They are used to store sequential numbers, such as invoice numbers or employee numbers, that are generated automatically by SQL Server. The value of the IDENTITY column uniquely identifies each row in a table.

Each table in a database can have a single IDENTITY column, of type *numeric* and scale zero. You can define an IDENTITY column when you create a table with a **create table** or **select into** statement, or add it later with an **alter table** statement.

By definition, IDENTITY columns cannot be updated and do not allow nulls. Each time you insert a row into a table, SQL Server automatically supplies a unique, sequential value for its IDENTITY column, beginning with the value "1". Manual insertions, deletions, transaction rollbacks, and server failures can create gaps in IDENTITY column values.

Defining an IDENTITY Column in a New Table

You can define an IDENTITY column with any desired precision—from 1 to 38 digits—in a new table. Specify the **identity** keyword, instead of **null** or **not null**, in the **create table** statement:

```
create table table_name
    (column_name numeric(precision,0) identity)
```

The following example creates a new table, *my_table*, with an IDENTITY column called *id_col*. Values for this column can range from 1 through 10^4 - 1, or 9,999:

```
create table my_table
    (id_col numeric(4,0) identity, col2 char(5))
```

Adding an IDENTITY Column to an Existing Table

To add an IDENTITY column to an existing table, specify the **identity** keyword in the **alter table** statement:

```
alter table table_name add column_name
    numeric(precision,0) identity
```

The following example adds an IDENTITY column, *row_id*, to the *stores* table:

```
alter table stores add row_id numeric(5,0) identity
```

When you add an IDENTITY column to a table, SQL Server assigns a unique sequential value, beginning with the value 1, to each existing row. If the table contains a large number of rows, this process can be time consuming. If the number of rows exceeds the maximum value allowed for the column (in this case, 99,999), the **alter table** statement fails.

Creating "Hidden" IDENTITY Columns Automatically

System Administrators can use the **auto identity** database option to automatically include a 10-digit IDENTITY column in new tables. To turn this feature on in a database, use:

```
sp_dboption database_name, "auto identity", "true"
```

Each time a user creates a new table without specifying either a **primary** key, a **unique** constraint, or an IDENTITY column, SQL Server automatically defines an IDENTITY column. The IDENTITY column is not visible when you use **select *** to retrieve all columns from the table. You must explicitly include the column name, **SYB_IDENTITY_COL** (all capital letters), in the select list.

Referencing an IDENTITY Column with syb_identity

Once you have created an IDENTITY column, you do not need to remember its name. You can use the **syb_identity** keyword, qualified by the table name where necessary, to reference the IDENTITY column in a **select**, **insert**, **update**, or **delete** statement.

For example, the following **update** statement finds the row in which the IDENTITY column equals 1 and changes the name of the store to Barney's:

```
update stores_cal
set stor_name = "Barney's"
where syb_identity = 1
```

Selecting an Existing IDENTITY Column into a New Table

To select an existing IDENTITY column into a new table, include the column name (or the **syb_identity** keyword) in the **select** statement's *column_list*:

```
select column_list
into table_name
from table_name
```

The following example creates a new table, *stores_cal_pay30*, based on columns from the *stores* table:

```
select row_id, stor_id, stor_name
into stores_cal_pay30
from stores_cal
where payterms = "Net 30"
```

When the New Column Doesn't Inherit the IDENTITY Property

When you select an IDENTITY column into a new table, the new column inherits the IDENTITY property unless one of the following conditions would result:

- A table with more than one IDENTITY column
- An IDENTITY column whose value must be computed
- An IDENTITY column value that cannot be guaranteed as unique

Selecting the IDENTITY Column More Than Once

A table cannot have more than one IDENTITY column. If an IDENTITY column is selected more than once, it is defined as NOT NULL in the new table. It does not inherit the IDENTITY property.

In the following example, the *row_id* column, which is selected once by name and once by the **syb_identity** keyword, is defined as NOT NULL in *stores_cal_pay60:*

```
select syb_identity, row_id, stor_id, stor_name
into stores_cal_pay60
from stores_cal
where payterms = “Net 60”
```

Defining a Column Whose Value Must Be Computed

IDENTITY column values are generated by SQL Server. New columns that are based on IDENTITY columns, but whose values must be computed rather than generated, cannot inherit the IDENTITY property.

If a table's **select** statement includes an IDENTITY column as part of an expression, the resulting column value must be computed. The new column is created as NULL if any column in the expression allows nulls, and NOT NULL otherwise.

In the following example, the *new_id* column, which is computed by adding 1000 to the value of *row_id*, is created NOT NULL:

```
select new_id = row_id + 1000, stor_name
into new_stores
from stores_cal
```

Column values are also computed if the select statement contains a **group by** clause or aggregate function. If the IDENTITY column is the argument of the aggregate function, the resulting column is created NULL. Otherwise it is created NOT NULL.

Defining a Column Whose Value May Not Be Unique

The value of the IDENTITY column uniquely identifies each row in a table. If a table's select statement contains a union or join, however, individual rows can appear multiple times in the result set.

An IDENTITY column that is selected into a table with a union or join does not retain the IDENTITY property. If the table contains the union of the IDENTITY column and a NULL column, the new column is defined as NULL. Otherwise, it is defined as NOT NULL.

Adding a New IDENTITY Column with select into

To define a new IDENTITY column in a table's **select into** statement, add a description of the IDENTITY column before the **into** clause. Note that the description includes the column precision, but not its scale:

```
select column_list
identity_column_name = identity(precision)
into table_name
from table_name
```

The following example creates a new table, *new_discounts*, from the *discounts* table and adds a new IDENTITY column, *id_col*:

```
select *, id_col=identity(5)
into new_discounts
from discounts
```

No table can have more than one IDENTITY column. If the *column_list* includes an existing IDENTITY column, and you add a description of a new IDENTITY column, the **select into** statement fails.

Using sp_help to Determine Whether a Column Has the IDENTITY Property

To determine whether a column has the IDENTITY property, use the **sp_help** system procedure on the column's base table. This procedure lists each column in the table. Columns with the IDENTITY property have an "Identity" value of 1; others, an "Identity" value of 0.

Here is the result of running **sp_help** on the *publishers* table:

```
Name                           Owner       Type
-------------------------      ----------- --------
publisher                      dbo         user table

Data_located_on_segment            When_created
-------------------------------    --------------------
default                            Jan  1 1900 12:00AM

Column_name  Type    Length  Nulls  Default_name Rule_name  Identity
----------   ------- ------  -----  ------------ ---------- --------
pub_id       char         4      0  NULL         pub_idrule        0
pub_name     varchar     40      1  NULL         NULL              0
city         varchar     20      1  NULL         NULL              0
state        char         2      1  NULL         NULL              0

index_name        index_description                        index_keys
--------------    ---------------------------------------  -------
pubind            clustered, unique located on default     pub_id

(1 row affected)
keytype    object      related_object  object_keys related_keys
-------    ------      -------------- ------------ ------------
primary    publishers  -- none --     pub_id, *, *, *, *, *, *, *
*, *, *, *, *, *, *, *, *
foreign    titles      publishers     pub_id, *, *, *, *, *, *, *
pub_id, *, *, *, *, *, *, *

(return status = 0)
```

Automatically Generating IDENTITY Column Values

The first time you insert a row into a table, SQL Server assigns the IDENTITY column a value of 1. Each new row you insert gets an IDENTITY column value one higher than the last. Server failures and transaction rollbacks can lead to gaps in these values.

Because SQL Server automatically generates the IDENTITY column value, **insert** statements should not list the IDENTITY column (or the **syb_identity** keyword) or specify its value.

Following are two examples of **insert** statements for a table with an IDENTITY column. The first statement has no column list:

```
insert stores_wash
values ("Best Books", "Net 60")
```

The second statement includes a column list, but omits the IDENTITY column, *row_id*:

```
insert stores_wash (stor_name, payterms)
values ("Books And Stuff", "Net 30")
```

As it executes each statement, SQL Server automatically supplies a value for the *row_id* column:

```
select * from stores_wash
```

```
row_id  stor_name                                payterms
------- ---------------------------------------- ------------
     1  Doc-U-Mat: Quality Laundry and Books     Net 60
     2  Eric the Read Books                      Net 60
     3  Best Books                               Net 60
     4  Books And Stuff                          Net 30
```

Allowing Explicit Inserts by Setting identity_insert on

There may be times when you want to insert an explicit value into an IDENTITY column. For example, you may want the first value inserted into the table to have an IDENTITY value of 101, rather than 1. Or you may need to restore a row that was deleted in error.

Only the table owner, the Database Owner, or System Administrator can explicitly insert a value into an IDENTITY column. Before inserting the data, the user must **set identity_insert on** for the column's base table. At any given time, you can **set identity_insert on** for one table in a database.

When **identity_insert** is turned **on**, each insert statement for the table must include a column list. The values list must specify an explicit IDENTITY column value.

The following example enables explicit inserts to the *sales_daily* table and specifies a starting value of 101 for the IDENTITY column:

```
set identity_insert sales_daily on
insert into sales_daily (syb_identity, stor_id)
values (101, “13-J-9”)
```

Unless you have created a unique index on the IDENTITY column, SQL Server does not verify the value's uniqueness before inserting it. You can insert any positive integer that falls within the column's range. This can cause gaps in IDENTITY column values.

Retrieving IDENTITY Column Values with @@*identity*

Use the @@*identity* global variable to retrieve the last value inserted into an IDENTITY column. The value of @@*identity* changes each time an **insert, select into**, or **bcp** statement inserts a row into a table.

- If the statement affects a table without an IDENTITY column, *@@identity* is set to 0.
- If the statement inserts multiple rows, *@@identity* reflects the last value inserted into the IDENTITY column.

This change is permanent. *@@identity* does not revert to its previous value if the **insert, select into**, or **bcp** statement fails, or if the transaction that contains it is rolled back.

Reaching the Column's Maximum Value

The maximum value that can be inserted into an IDENTITY column is $10^{precision} - 1$. If you do not specify a precision for the IDENTITY column, SQL Server uses the default precision for *numeric* columns (18 digits).

Once an IDENTITY column reaches its maximum value, all further **insert** statements return an error that aborts the current transaction.When this happens, use the **create table** statement to create a new table identical to the old, but with a larger precision for the IDENTITY column. Once you have created the new table, use the **insert** statement or the **bcp** utility to copy the data from the old table to the new.

Bulk Copying a Table with an IDENTITY Column

By default, when you bulk copy data into a table with an IDENTITY column, **bcp** assigns each row a temporary IDENTITY column value of 0. As it inserts each row into the table, the server assigns it a unique, sequential IDENTITY column value, beginning with the value 1. To enter an explicit IDENTITY column value for each row, specify the **-E**(UNIX) or **/identity** (OpenVMS) flag.

By default, when you bulk copy data from a table with an IDENTITY column, **bcp** excludes all information about the column from the output file. To copy the existing IDENTITY column values into the output file, specify the **-E** (UNIX) or **/identity** (OpenVMS) flag.

Including an IDENTITY Column in a View

You can define a view that includes an IDENTITY column by listing the column name, or the **syb_identity** keyword, in the view's *select_statement*. You cannot add a new IDENTITY column to a view with the *identity_column_name* **= identity(***precision***)** syntax.

The underlying column retains the IDENTITY property. When you update a row through the view, you cannot specify a new value for the IDENTITY column. When you insert a row through the view, SQL

Server automatically generates a new, sequential value for the IDENTITY column. Only the table owner, Database Owner, or System Administrator can explicitly insert a value into the IDENTITY column after turning **identity_insert on** for the column's base table. (Note: it is not sufficient to **set identity_insert on** for the view.)

Although the column behaves like an IDENTITY column, SQL Server does not always recognize that it has the IDENTITY property. A view is considered not to have an IDENTITY column if it:

- Joins columns from multiple tables or
- Includes an aggregate function or
- Selects an IDENTITY column more than once or
- Includes the IDENTITY column as part of an expression

Under these circumstances, you cannot use the **syb_identity** keyword to select a column from the view. If you execute the **sp_help** procedure on the view, all columns display an *Identity* value of 0, indicating that there is no IDENTITY column.

In the following example, the *row_id* column is not recognized as an IDENTITY column with respect to the *store_discount*s view because *store_discounts* joins columns from two tables:

```
create view store_discounts
as
select stor_name, row_id, discount
from stores, new_discounts
where stores.stor_id = new_discounts.stor_id
```

Creating a User-Defined Datatype with the IDENTITY Property

You can use the **sp_addtype** system procedure to create a user-defined datatype with the IDENTITY property. The new type must be based on a physical type of *numeric* with a scale of zero:

```
sp_addtype typename, "numeric (precision , 0)",
"identity"
```

The following example creates a user-defined type, *IdentType*, with the IDENTITY property:

```
sp_addtype IdentType, "numeric(4,0)", "identity"
```

When you create a column from an IDENTITY type, you can specify either **identity** or **not null**—or neither one—in the **create** or **alter table** statement. The column automatically inherits the IDENTITY property.

Following are three different ways to create an IDENTITY column from the *IdentType* user-defined type:

```
create table new_table (id_col IdentType)
create table new_table (id_col IdentType identity)
create table new_table (id_col IdentType not null)
```

➤ ***Note***

If you try to create a *null* column from an IDENTITY type, the **create** or **alter table** statement fails.

Creating IDENTITY Columns from Other User-Defined Datatypes

You can also create IDENTITY columns from user-defined datatypes that do not have the IDENTITY property. The user-defined types must have a physical datatype of *numeric*, a scale of zero, and must be defined as **not null**.

Controlling Gaps in IDENTITY Column Values

IDENTITY column values can range from a low of 1 to a high of $10^{\text{column precision}} - 1$. SQL Server divides the set of possible values into blocks of consecutive numbers, and makes one block of numbers at a time available in memory.

When assigning an IDENTITY column value, SQL Server draws the next available value from the block. Once all the values in the block have been used, SQL Server makes the next block available. Choosing the next IDENTITY value from a block of available numbers improves server performance, but can lead to gaps in column values.

Gaps Due to Server Failures

Whenever SQL Server fails, it discards any remaining values in the current block. When you restart SQL Server, it makes the next block of numbers available for the IDENTITY column.

For example, a two-digit IDENTITY column can have values ranging from 1–99. SQL Server might make values 1–5 available for the first set of insertions:

Potential IDENTITY Column Values	
1 - 5	6 - 99
available	not yet available

- The first time you insert a row, SQL Server assigns the IDENTITY column a value of 1.
- The second time you insert a row, SQL Server assigns a value of 2.
- If SQL Server fails at this point, it discards the remaining numbers in the block (3, 4, and 5).
- When you restart SQL Server, it makes a new block of numbers (6 through 10) available:

Potential IDENTITY Column Values		
1 - 5	6 - 10	11 - 99
no longer available	available	not yet available

- The next time you insert a row, SQL Server assigns the IDENTITY column a value of 6. The values 3, 4, and 5 are skipped.

You use the **identity burning set factor** configuration variable to control the size of gaps resulting from server failure. This variable, which is set during installation, determines what percentage of the potential column values is contained in each block of available numbers. The default value, 5000, releases .05% of potential column values for use at a time.

System Administrators can reset the **identity burning set factor** variable using the **-y** option of **buildmaster** or the **sp_configure** system procedure:

```
sp_configure "identity burning set factor", value
```

Determine what percentage of available numbers you want to make available at a time. This number should be high enough for good performance, but not so high that gaps are unacceptably large. Express the number in decimal form, then multiply it by 10,000,000 (10 raised to the power 7) to get the correct value for **sp_configure**.

For example, to release 15% (.15) of the potential IDENTITY column values at a time, you specify a *value* of .15 times 10^7 (or 1,500,000) in **sp_configure**:

```
sp_configure "identity burning set factor", 1500000
```

Then execute the **reconfigure** command to implement the change.

Gaps Due to Insertions, Deletions, and Rollbacks

Manual insertions into the IDENTITY column, deletion of rows, and transaction rollbacks can all create gaps in IDENTITY column values. These gaps are not affected by the setting of the **identity burning set factor** configuration variable.

For example, assume you have an IDENTITY column with the following values:

```
select syb_identity from stores_cal

 id_col
 -------
      1
      2
      3
      4
      5

(5 rows affected)
```

You can delete all rows for which the IDENTITY column falls between 2 and 4, leaving gaps in the column values:

```
delete stores_cal
where syb_identity between 2 and 4

select syb_identity from stores_cal

 id_col
 -------
      1
      5
```

After turning **identity_insert** on for the table, the table owner, Database Owner, or System Administrator can manually insert any legal value greater than 5. For example, inserting a value of 55 would create a large gap in IDENTITY column values:

```
insert stores_cal
(syb_identity, stor_id, stor_name)
values (55, “5025”, “Good Reads”)

select syb_identity from stores_cal

id_col
 -------
      1
      5
     55
```

If **identity_insert** is then turned off, SQL Server assigns an IDENTITY column value of 55 + 1, or 56, for the next insertion. If the transaction that contains the **insert** statement is rolled back, SQL Server discards the value 56 and uses a value of 57 for the next insertion.

Standards and Compliance

Standard	Compliance Level
SQL89	Transact-SQL extension
SQL92	Transact-SQL extension

See Also

Commands	**alter table**, **create table**, **delete**, **insert**, **select**, **set**, **update**, **where** Clause
Topics	Datatypes, Null Values
System procedures	**sp_dboption**

Joins

Function

Joins compare two or more tables (or views) by specifying a column from each, comparing the values in those columns row by row, and concatenating rows that have matching values. Joins can also be stated as subqueries. (See "Subqueries" for more information.)

Join Syntax

A join can be embedded in a **select**, **update**, **insert**, **delete**, or *subquery*. Other search conditions and clauses may follow the join condition(s). Joins use the following syntax:

```
Start of select, update, insert, delete, or subquery
   from {table_list | view_list}
   where [not]
       [table_name.| view_name.]column_name
           join_operator
       [table_name. | view_name.]column_name
   [{and | or} [not]
       [table_name.|view_name.]column_name
           join_operator
       [table_name.|view_name.]column_name]...

End of select, update, insert, delete, or subquery
```

Specifying Which Tables and Views to Join

Use **from** to specify which tables and views to join. All tables and/or views referenced elsewhere in the statement must be included in the **from** clause.

At most, a query can reference 16 tables. This maximum includes:

- Tables (or views on tables) listed in the **from** clause
- Each instance of multiple references to the same table (self-joins)
- Tables referenced in subqueries
- Work tables created as a result of the query

The following example joins columns from the tiles and publishers tables, doubling the price of all books published in California:

```
update titles
  set price = price * 2
  from titles, publishers
  where titles.pub_id = publishers.pub_id
  and publishers.state = "CA"
```

Determining Which Rows to Include in the Results

Use **where** to determine which rows are included in the results. **where** specifies the connection between the table(s) and view(s) named in the **from** clause. Be sure to qualify column names if there is ambiguity about the table or view to which they belong.

Joined Columns Must Have Comparable Datatypes

The columns being joined must have the same or comparable datatypes. Use the **convert** function when comparing columns whose datatypes cannot be implicitly converted. Joins cannot be used for columns containing *text* or *image* values.

Comparison Operators

Joins based on a comparison of scalar values are called **theta joins.** Theta joins use the following comparison operators:

Symbol	Meaning
=	equal to
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
!=	not equal to
<>	not equal to
!>	not greater than
!<	not less than

Table 3-18: Comparison operators

Connecting Join Expressions

You can use more than one join operator to join more than two tables or to join more than two pairs of columns. These "join expressions" are usually connected with **and**, although **or** is also legal.

Following are two examples of joins connected by **and**. The first lists information about books (type of book, author, and title), ordered by book type. Books with more than one author have multiple listings, one for each author.

```
select type, au_lname, au_fname, title
from authors, titles, titleauthor
where authors.au_id = titleauthor.au_id
and titles.title_id = titleauthor.title_id
order by type
```

The second finds the names of authors and publishers that are located in the same city and state:

```
select au_fname, au_lname, pub_name
from authors, publishers
where authors.city = publishers.city
and authors.state = publishers.state
```

Using the not Operator

The expression:

not *column_name* = *column_name*

is equivalent to:

column_name* != *column_name

Equijoins

Joins based on equality (**=**) are called **equijoins**. A natural join is an equijoin with only one of the matching columns displayed in the results. For example, the following join finds the names of authors and publishers that are located in the same city:

```
select au_fname, au_lname, pub_name
from authors, publishers
where authors.city = publishers.city
```

Not Equal Joins

The **not equal join** (!= or <>) is usually best used with a self-join. For each book in *titles*, the following example finds all other books of the same type that have a different price:

```
select titles.type, t1.title_id, t1.price,
t2.title_id, t2.price
from titles t1, titles t2
where t1.type = t2.type
and t1.price != t2.price
```

Exercise caution when interpreting the results of a not equal join. For example, it would be easy to think you could find the authors that do not live in a city in which a publisher is located with a not equal join:

```
/*Incorrect Statement*/
   select distinct au_lname, authors.city
   from publishers, authors
   where publishers.city != authors.city
```

However, this query actually finds the authors who live in a city where *some* publisher is not located—which is all of them. The correct SQL statement is a subquery:

```
select distinct au_lname, authors.city
from publishers, authors
where authors.city not in
(select city from publishers
 where authors.city = publishers.city)
```

Self-Joins

Joins that compare values within the same column of one table are called **self-joins.** To distinguish the two roles in which the table appears, use aliases.

For example, the following join finds the authors with the same postal code as other authors, eliminating the rows where the author's name matches itself. It uses the aliases *au1* and *au2* for the *authors* table:

```
select au1.au_fname, au1.au_lname, au2.au_fname,
au2.au_lname
from authors au1, authors au2
where au1.postalcode = au2.postalcode
and (au1.au_lname + au1.au_fname) !=
    (au2.au_lname + au2.au_fname)
```

Outer Joins

Joins that include all rows regardless of whether there is a matching row are called **outer joins**. SQL Server supports both left and right outer joins.

The **left outer join**, ***=**, selects all rows from the first table that meet the statement's restrictions. The second table generates values if there is a match on the join condition. Otherwise, the second table generates null values.

For example, the following left outer join lists all authors and finds the publishers (if any) in their city:

```
select au_fname, au_lname, pub_name
from authors, publishers
where authors.city *= publishers.city
```

The **right outer join**, **=***, selects all rows from the second table that meet the statement's restrictions. The first table generates values if there is a match on the join condition. Otherwise, the first table generates null values.

A table is either an inner or an outer member of an outer join: if the join operator is ***=** the second table is the inner table; if the join operator is **=*** the first table is the inner table. You can compare a column from the inner table to a constant as well as using it in the outer join. For example, if you wanted to find out which *title* had sold more than 4,000 copies:

```
select qty, title from salesdetail, titles
where qty > 4000
and titles.title_id *= salesdetail.title_id
```

However, the inner table in an outer join cannot also participate in a regular join clause.

Comparison of Nulls

Null values in tables or views being joined will never match each other. Since *bit* columns do not permit NULLs, a value of "0" appears in an outer join when there is no match for a *bit* column that is in the inner table.

Join Views

If you define a view with an outer join, and then query the view with a qualification on a column from the inner table of the outer join, the results may be other than what you expect. All rows from the inner table are returned. Rows that do not meet the qualification show a NULL value in the appropriate columns of those rows.

The following rules determine what types of updates you can make to columns through join views:

- **delete** statements are not allowed on join views.
- **insert** statements are not allowed on join views created **with check option.**
- **update** statements are allowed on join views **with check option.** The update fails if any of the affected columns appears in the **where**

clause, in an expression that includes columns from more than one table.

- If you insert or update a row through a join view, all affected columns must belong to the same base table.

Using Joins in Cursor Definitions

If you use a join in a cursor definition, you cannot **delete** rows from the cursor result set, even if the cursor is updatable.

Standards and Compliance

Any statement that contains the join operators *= or =* is a Transact-SQL extension.

See Also

Commands	**delete**, **insert**, **select**, **update**, **where** Clause
Functions	Datatype Conversion Functions
Topics	Subqueries
System procedures	**sp_helpjoins**

Login Management

Function

Mechanisms used to verify the identities of SQL Server users and to administer SQL Server login accounts.

Login Management System Procedures

The following system procedures are used to manage SQL Server login accounts:

Procedure	Function
sp_addlogin	Adds new user accounts to SQL Server. It allows you to specify the login name, password, default database, default language, and the user's full name.
sp_addremotelogin	Authorizes an existing user on another SQL Server to execute remote procedure calls to this SQL Server.
sp_displaylogin	Displays information about SQL Server login accounts.
sp_droplogin	Drops a SQL Server login account. An account cannot be dropped if the owner is a user in any database on the server. It is recommended that accounts be locked rather than dropped.
sp_dropremotelogin	Drops a remote user's login.
sp_helpremotelogin	Reports information on a particular remote server's logins, or on all remote servers' logins.
sp_locklogin	Locks or unlocks a SQL Server account so that the user cannot log in. Locked accounts can own objects and can be Database Owners.
sp_modifylogin	Allows you to add or modify a SQL Server user's default database, default language, or full name.
sp_password	Changes a user's SQL Server password.
sp_serveroption	Displays or changes server options. The **net password encryption** option specifies that passwords are to be encrypted when sent across the network in remote procedure calls. The **timeouts** option disables the normal timeout code used by the local server, so the site connection handler does not automatically drop the physical connection after a minute with no logical connection.

Table 3-19: System procedures for managing SQL Server login accounts

Procedure	Function
sp_who	Displays information about a particular SQL Server user or all current SQL Server users and processes.

Table 3-19: System procedures for managing SQL Server login accounts (continued)

Password Management

Management of passwords is essential to the security of any system. Password secrecy in SQL Server is ultimately the responsibility of the individual user, but the following features help to keep passwords confidential:

- Passwords must be at least 6 bytes long. This is enforced by **sp_addlogin** and **sp_password**.
- User passwords are stored in *master..syslogins* in encrypted form, using an exportable one-way encryption algorithm.
- You can globally enforce password expiration using the **password expiration interval** option to **sp_configure**. This variable sets the number of days that passwords remain in effect after they are changed. Users are warned before their passwords expire. Warnings begin when the number of days remaining before expiration is less than 25% of the value of **password expiration interval** or seven days, whichever is greater. When a user's password has expired, the user can log into SQL Server but cannot execute any commands until he or she has changed the password using **sp_password**. See the *System Administration Guide* for more information.
- By setting the **net password encryption** option to **sp_serveroption,** you may choose to send either unencrypted passwords or client-side encrypted passwords across the network when executing remote procedure calls. Similarly, you can use the **isql** encryption option **(-X** for UNIX, **/encrypt** for OpenVMS) to establish password encryption between **isql** and the server. See **sp_serveroption** and the *SQL Server Utility Programs* for more information.
- If all System Security Officer passwords are lost or forgotten, there is an option to the **dataserver** command that generates a new password for a System Security Officer's account. See **dataserver** in the *SQL Server Utility Programs* for more information.

See Also

Topics	Roles
System procedures	**sp_addlogin, sp_addremotelogin, sp_configure, sp_droplogin, sp_dropremotelogin, sp_locklogin, sp_modifylogin, sp_password, sp_serveroption**
Utility programs	**dataserver, isql**

Null Values

Function

Nulls mark columns having an unknown value (as opposed to those that have 0 or blank as a value). NULL allows you to distinguish between a deliberate entry of zero (for numerical columns) or blank (for character columns) and a non-entry (NULL for both numerical and character columns).

Declaring a Column's Null Type

When you create a column, you specify either a null type or the IDENTITY property. The null type determines whether the column requires an explicit entry.

Creating a New Table

In **create table** statements, declare each columns as **null**, **not null**, or **identity** or accept the default null type for the database:

```
column_name datatype [null | not null | identity]
```

null indicates that no entry is required for the column. **not null** means that a user must provide a value for the column if no defined default value exists. By definition, columns with the IDENTITY property do not allow nulls.

The default null type for a column in a **create table** statement is to **not** allow nulls. To operate in ANSI "null mode," which requires that columns allow nulls by default, use **sp_dboption** to set the **allow nulls by default** option to **true**.

Adding a Column to an Existing Table

When adding a column to an existing table, you can specify it as **null** or **identity**:

```
column_name datatype [null | identity]
```

When you add a column other than an IDENTITY column, SQL Server sets each existing row to NULL

Displaying a Column's Null Type

The system procedure **sp_help** reports the *nulltype* of a column (1 or 0, depending on whether or not the column accepts null values).

Nulls Require Variable Length Datatypes

Only columns with variable-length datatypes can store null values. When you create a NULL column with a fixed-length datatype, SQL Server automatically converts it to the corresponding variable-length datatype. SQL Server does not inform you of the type change.

The following chart lists the fixed-length datatypes and the variable-length datatypes to which they are converted. Certain variable-length datatypes, such as *moneyn*, are reserved types; you cannot use them to create columns, variables, or parameters:

Original Fixed-Length Datatype	Converted To
char	*varchar*
nchar	*nvarchar*
binary	*varbinary*
datetime	*datetimn*
float	*floatn*
int, *smallint*, and *tinyint*	*intn*
decimal	*decimaln*
numeric	*numericn*
money and *smallmoney*	*moneyn*

Table 3-20: Conversion of fixed-length to variable-length datatypes

These changes affect the handling of *char*, *nchar*, and *binary* data. Data entered into these columns follow the rules for variable-length columns, rather than being padded with spaces or zeros to the full length of the column specification.

Testing a Column for Null Values

Use **is null** in **where**, **if**, and **while** clauses to compare column values to NULL and to select them or perform a particular action based on the results. Only columns that return a value of TRUE are selected or result in the specified action; those that return FALSE or UNKNOWN do not.

The following example selects only rows for which *advance* is $5000 or null:

```
select title_id, advance
from titles
where advance < $5000 or advance is null
```

Comparisons that Return TRUE

In general, the result of comparing null values is UNKNOWN, since it is not possible to determine if NULL is equal (or not equal) to a given value or to another NULL. There are three exceptions. The following cases return TRUE when *expression* is any column, variable or literal, or combination of these, which evaluates as NULL:

- *expression* **is [not] null**
- *expression* = **null** and *expression* != **null**
- *expression* = @x and *expression* != @x, where @x is a variable or parameter containing NULL. This exception facilitates writing stored procedures with null default parameters.

Note that the right-hand side of all of these exceptions is a literal null, or a variable or parameter containing NULL. If the right hand side of the comparison is an expression (such as @nullvar + 1), the entire expression evaluates to NULL.

Following these rules, null column values do not join with other null column values. Comparing null column values to other null column values in a **where** clause always returns UNKNOWN for null values, regardless of the comparison operator, and the rows are not included in the results. For example, this query returns no result rows where *column1* contains NULL in both tables (although it may return other rows):

```
select column1
from table1, table2
where table1.column1 = table2.column1
```

Difference Between FALSE and UNKNOWN

Although neither FALSE nor UNKNOWN returns values, there is an important logical difference between FALSE and UNKNOWN, because the opposite of false ("not false") is true. For example, "1 = 2" evaluates to false, and its opposite, "1 != 2", to true. But "not unknown" is still unknown. If null values are included in a comparison, you can't negate the expression to get the opposite set of rows or opposite truth value.

Inserting a Null Value into a Column

You can explicitly insert NULL into a column:

```
values({expression | null}
    [, {expression | null}]...)
```

The following example shows two equivalent **insert** statements. In the first statement, the user explicitly inserts a NULL into column *t1*. In the second, SQL Server provides a NULL value for *t1* because the user has not specified an explicit column value:

```
create table test
(t1 char(10) null, t2 char(10) not null)

insert test
values (null, "stuff")

insert test (t2)
values ("stuff")
```

Changing a Column's Value to Null

Use the **update** statement to set a column value to NULL. It's syntax is:

```
set column_name = {expression | null}
    [, column_name = {expression | null}]...
```

The following example finds all rows in which the *title_id* is TC3218 and replaces the *advance* with NULL.

```
update titles
set advance = null
where title_id = "TC3218"
```

Declaring NULL as the Default for a Parameter

In the **create procedure** statement, you can declare NULL as the default value for individual parameters:

```
create procedure procedure_name
    @param datatype [ = null ]
    [, @param datatype [ = null ]]...
```

The following example creates a stored procedure, and assigns the parameter *@id* a default value of NULL:

```
create procedure myproc @id varchar(6) = null
as
select title_id, pub_id from titles
where title_id = @id
```

For information on NULL as a parameter default, see "**create procedure**".

Substituting a Value for Nulls

Use the **isnull** built-in function to substitute a particular value for nulls. The substitution is made only for display purposes; actual column values are not affected. The syntax is:

```
isnull(expression, value)
```

For example, use the following statement to select all the rows from *test*, and display all the null values in column *t1* with the value unknown.

```
select isnull(t1, "unknown")
from test
```

If you use **is null** or "= NULL" to try to find null values in columns defined as **not null**, SQL Server generates an error message.

Rules and Null Values

It is not possible to define a column to allow nulls, and then to override this definition with a rule that prohibits null values. For example, if a column definition specifies NULL and the rule specifies this:

```
@val in (1,2,3)
```

an implicit or explicit NULL does not violate the rule. The column definition overrides the rule, even a rule that specifies:

```
@val is not null
```

Defaults and Null Values

If you specify NOT NULL when you create a column and do not create a default for it, an error message occurs when a user fails to make an entry in that column during an insert. In addition, the user cannot **insert** or **update** such a column with NULL as a value. The following table illustrates the result of the relationship between the existence of a default and the definition of a column as NULL or NOT NULL The three possible results are: a null value for the column, the default value for the column, or an error message.

Column Definition	No Entry, No Default	No Entry, Default	Enter NULL, No Default	Enter NULL, Default
NULL	null value	default value	null value	null value
NOT NULL	error	default value	error	error

Table 3-21: Column definition and null defaults

"No Entry" means that the **insert** statement has a column list which does not include the column.

text and image Columns

text and *image* columns created with **insert** and NULL are not initialized and contain no value. They do not use storage space and cannot be accessed with **readtext** or **writetext**. However, when a NULL value is written in a *text* or *image* column with **update**, the column is initialized, a valid text pointer to the column is inserted into the table, and a 2K data page is allocated to the column. Once the column is initialized, it can be accessed by **readtext** and **writetext**. See ''text and image Datatypes'' for more information.

Using the "null" Character String

Only columns for which NULL was specified in the **create table** statement and into which you have explicitly entered NULL (no quotes), or into which no data has been entered, contain null values. Avoid entering the character string "NULL" (with quotes) as data for a character column. It can only lead to confusion. Use "N/A," "none," or the like instead. When you want to enter the value NULL explicitly, do **not** use single or double quotes.

Nulls versus the Empty String

The empty string (" "or ' ') is always stored as a single space in variables and column data. This concatenation statement:

```
"abc" + "" + "def"
```

is equivalent to "abc def", not to "abcdef". The empty string is never evaluated as NULL.

Aggregates and Nulls

By default, the **ansinull** option is set **off**. Aggregates ignore null values, except **count**(*), which includes them. For example, in the calculation **avg***(advance)*, null values in the *advance* column are not counted, either in calculating the total amount or the number of values.

For SQL92 compliance, **set ansinull on**. Aggregates that encounter nulls generate a warning.

group by Clauses and Nulls

When the **group by** clause is used, null values form their own group.

order by Clauses and Nulls

With **order by**, null values come before all others.

select distinct and Nulls

In a **select** clause with the keyword **distinct**, which only selects unique rows, null values are considered duplicates of each other. Only one NULL is selected, no matter how many are encountered.

Subqueries That Return No Values

The result of a subquery that returns no values is NULL. If a subquery returns NULL, the query failed.

Inserting Nulls into Columns That Do Not Allow Them

To insert data with **select** from a table that has null values in some fields into a table that does not allow null values, you must provide a substitute value for any NULL entries in the original table. For example, to insert data into an *advances* table that won't allow null values, this example substitutes "0" for the NULL fields:

```
insert advances
select pub_id, isnull(advance, 0) from titles
```

Without the **isnull** function, this command would insert all the rows with non-null values into *advances*, and produce error messages for all the rows where the *advance* column in *titles* contained NULL.

If this kind of substitution cannot be made for your data, it is not possible to insert the data containing null values into the columns with the NOT NULL specification.

Expressions That Evaluate to Null

An expression with an arithmetic or bitwise operator evaluates to NULL if any of the operands are null. For example:

```
1 + column1
```

evaluates to NULL if *column1* is NULL.

See Also

Commands	**create procedure**, **create table**, **group by** and **having** Clauses, **insert**, **select**, **update**
Functions	Aggregate Functions
Topics	Subqueries, text and image Datatypes
System procedures	**sp_help**

Parameters

Function

Parameters are arguments to a stored procedure. You define parameters when you create the procedure and supply their values when you execute the procedure. Not all procedures require parameters.

Declaring Procedure Parameters

You declare procedure parameters in the **create procedur**e statement. The declaration must include the parameter name and datatype and can include other optional information:

```
create procedure [owner.]procedure_name[;number]
   [ [(] @parameter_name datatype [= default] [output]
       [,@parameter_name datatype [= default] [output]]
       ... [)] ]
   [with recompile]
   as SQL_statements
```

Parameter Names

Parameter names must be preceded by the "@" symbol and conform to the rules for identifiers. Enclose any parameter value that includes punctuation (such as an object name qualified by a database name or owner name) in single or double quotes.

Parameter names are local to the procedure; the same parameter names can be used in multiple procedures.

Parameter Datatypes

Parameters can have any datatype except *text* or *image*. Some datatypes expect a length or precision and scale in parentheses. See the "Datatypes" section for a list of SQL Server-supplied datatypes and their syntax.

Default Parameter Values

To make a parameter optional, specify a default value. Users can execute the procedure without specifying an explicit value for that parameter.

The default must be a constant. It can include wildcard characters (%, _, [], and [^]) if the procedure uses the parameter name with the keyword **like**.

The default can be NULL. The procedure definition can specify that some action be taken if the parameter value is NULL. See **create procedure** for examples.

Output Parameters

Use the **output** keyword for return parameters. When the procedure is executed, these parameters return their values to the calling procedure or batch. See **execute** for more information.

Specifying Parameter Values

You specify parameter values when you execute the stored procedure:

```
[execute] [@return_status = ]
   [[server.]database.]owner.]procedure_name[;number]
   [[@parameter_name =] value |
       [@parameter_name =] @variable [output]
   [,[@parameter_name =] value |
       [@parameter_name =] @variable [output]... ]]
   [with recompile]
```

The value can be a constant or the name of a database object such as a table or column. It cannot be of type *text* or *image*.

Specifying Values by Name

You can specify parameter values in any order using the *parameter_name = value* syntax. If a parameter has a default, you do not need to specify its value.

The following procedure shows the datatype of any column. Here, the procedure displays the datatype of the *qty* column from the *salesdetail* table.

```
create procedure showtype @tabname varchar(18),
@colname varchar(18) as
  select syscolumns.name, syscolumns.length,
  systypes.name
  from syscolumns, systypes, sysobjects
  where sysobjects.id = syscolumns.id
  and @tabname = sysobjects.name
  and @colname = syscolumns.name
  and syscolumns.type = systypes.type
```

When the procedure is executed, the *@tabname* and *@colname* values can be given in a different order from the **create procedure** statement if they are specified by name:

```
exec showtype
@colname = qty , @tabname = salesdetail
```

Specifying Values by Order

If you omit the parameter names, you must supply them in the order in which they were defined in the **create procedur**e statement. Use commas between parameter values. In the following example, the values for *@tabname* and *@colname* are specified without names but in the same order as in the **create procedure** statement:

```
exec showtype salesdetail, qty
```

If a parameter has a default, you do not need to specify its value. Use a comma for each optional parameter whose value you do not specify.

Returning an Execution Status

Use *@return_status* = to return an execution status to the calling procedure or batch. The execution status can be a SQL Server-supplied value or a value supplied by the user in a **return** command.

Standards and Compliance

Stored procedures and their parameters are Transact- SQL extensions.

See Also

Commands	**create procedure, declare, drop procedure, execute**
Topics	Datatypes, Variables (Local and Global)

Roles

Function

Roles provide individual accountability for users performing system administration and security-related tasks in SQL Server.

About Roles

Individual server login accounts can be granted specific roles. When a user with the System Administrator role logs in, he or she can perform system administration tasks that can then be audited and attributed to the individual login.

The roles are:

- System Administrator (**sa_role**), required for system administration tasks such as:
  - Installing SQL Server
  - Managing disk storage
  - Granting permissions to SQL Server users
  - Transferring bulk data between SQL Server and other software programs
  - Modifying, dropping, and locking server login accounts
  - Monitoring SQL Server's automatic recovery procedure
  - Diagnosing system problems and reporting them as appropriate
  - Fine-tuning SQL Server by changing the configurable system parameters
  - Creating user databases and granting ownership of them
  - Granting and revoking the System Administrator role
  - Setting up groups (which are convenient in granting and revoking permissions)

  SQL Server does no permission checking on objects accessed by System Administrators. In addition, a System Administrator becomes the Database Owner in any database he or she is using by assuming user ID 1.
- System Security Officer (**sso_role**), required for security-related tasks such as:
  - Creating server login accounts

- Granting and revoking the System Security Officer and Operator roles
- Changing the password of any account
- Setting the password expiration interval
- Managing the audit system

A System Security Officer is not exempt from permission checking, and is not automatically treated as Database Owner in any database he or she is using.

- Operator (**oper_role**), used to back up and restore databases on a server-wide basis. These operations can be performed in a single database by the Database Owner, or by a System Administrator. The Operator role allows a single user to back up and restore multiple databases without having to be a user in each one.

More than one login account on a SQL Server can be granted any role, and one login can be granted more than one role.

The system procedures **sp_droplogin**, **sp_locklogin**, and **sp_role** ensure that there is always at least one unlocked login account that has been granted the System Security Officer role, and one that has been granted the System Administrator role. You cannot lock or drop the last remaining System Security Officer or System Administrator login.

Role Management

Granting and Revoking Roles to Users

Roles are granted to and revoked from individual users with the **sp_role** system procedure. Only a System Administrator can grant the System Administrator role, and only a System Security Officer can grant the System Security Officer and Operator roles.

Roles and Stored Procedures

You can use the **grant execute** command to grant execute permission on a stored procedure to all users who have been granted a specified role. Similarly, **revoke execute** removes this permission.

However, **grant execute** permission does not prevent users who do not have the specified role from being granted execute permission on a stored procedure. If you want to ensure, for example, that all users who are not System Administrators can never be granted permission to execute a stored procedure, you can use the **proc_role** system function within the stored procedure itself. It checks to see whether

the invoking user has the correct role to execute the procedure. See "System Functions" for more information.

Enabling and Disabling Roles in a Session

The **role** option of the **set** command allows you to turn a specified role on or off for your current session. When you log in, all roles that have been granted to you are enabled. If, for example, you have been granted the System Administrator role, you assume the identity of Database Owner in any database you use. You may want to disable your System Administrator role at times and assume your "real" user identity instead, using this command:

```
set role "sa_role" off
```

Displaying Role Information

- The **sp_displaylogin** system procedure displays information about a server login, including server user ID, name, roles granted to the login, date of last password change, and whether the login is locked.
- The **show_role** system function returns the user's current active roles, if any. See "System Functions" for more information.

Commands Requiring Specific Roles

The following tables list SQL commands and system procedures that require the user who is executing them to have a particular role (System Administrator, System Security Officer, or Operator) or status (Database Owner, table owner).

SQL Commands that Require Roles

Commands	Required Role or User Status
alter database	SA or Database Owner
create database	SA or permission granted with **grant**
Some **dbcc** commands	SA, Database Owner, table owner
disk init **disk refit** **disk reinit**	SA

Table 3-22: Roles required for SQL commands

Commands	Required Role or User Status
disk mirror **disk remirror** **disk unmirror**	SA
drop database	SA or Database Owner
dump database **dump transaction**	Operator or Database Owner
grant all **grant create database** **grant with grant option**	SA or object owner SA SA or object owner
kill	SA
load database **load transaction**	Operator or Database Owner
reconfigure	SA
revoke all **revoke create database**	SA or object owner SA
setuser	SA or Database Owner
shutdown	SA
update statistics	SA or table owner

Table 3-22: Roles required for SQL commands (continued)

System Procedures that Require Specific Roles

System Procedures	Required Role or User Status
sp_addalias	SA or Database Owner
sp_addgroup	SA or Database Owner
sp_addlanguage	SA
sp_addlogin	SSO
sp_addremotelogin	SA
sp_addsegment	SA or Database Owner
sp_addserver	SSO
sp_addumpdevice	SA
sp_adduser	SA or Database Owner
sp_auditdatabase	SSO

Table 3-23: Roles required for system procedures

System Procedures	Required Role or User Status
sp_auditobject	SSO
sp_auditoption	SSO
sp_auditsproc	SSO
sp_auditlogin	SSO
sp_changedbowner	SA
sp_changegroup	SA or Database Owner
sp_clearstats	SA
sp_configure	
without parameters	any user
general updates options	SA
allow updates option	SSO
audit queue size option	SSO
passwordexp option	SSO
sp_dboption	
without parameters	any user
with parameters	SA or Database Owner
sp_diskdefault	SA
sp_displaylogin	SA, SSO, or account owner
sp_dropalias	SA or Database Owner
sp_dropdevice	SA
sp_dropgroup	SA or Database Owner
sp_droplanguage	SA
sp_droplogin	SA
sp_dropmessage	SA, Database Owner, or object owner
sp_dropremotelogin	SA
sp_dropsegment	SA or Database Owner
sp_dropserver	SSO
sp_dropuser	SA or Database Owner
sp_extendsegment	SA or Database Owner
sp_locklogin	SSO or SA
sp_logdevice	SA or Database Owner
sp_modifylogin	SA
sp_monitor	SA

Table 3-23: Roles required for system procedures (continued)

System Procedures	Required Role or User Status
sp_password	SSO or account owner
sp_placeobject	SA, Database Owner, table owner
sp_remoteoption with parameters	SSO
sp_rename	SA, Database Owner, object owner
sp_renamedb	SA
sp_reportstats	SA
sp_role	SSO for SSO and Oper. roles; SA for SA role
sp_serveroption **net password encryption** option **timeouts** option	 SSO SA
sp_setlangalias	SA
sp_sortorder	SA

Table 3-23: Roles required for system procedures (continued)

See Also

Commands	**grant, revoke, set**
Functions	System Functions
System procedures	**sp_configure, sp_displaylogin, sp_locklogin, sp_role**

Search Conditions

Function

Search condition set the conditions in a **where** or **having** clause. (Joins and subqueries are specified in the search conditions: see the "Joins" and "Subqueries" sections for full details.)

Using Search Conditions

Search conditions immediately follow the keywords **where** or **having** in a **select, insert, update,** or **delete** statement. A **where** clause can include a maximum of 128 search conditions per table.

Comparing where *and* having

having search conditions differ from **where** search conditions only in that aggregate functions are not allowed in **where** clauses:

```
having avg(price) > $20
```

is legal;

```
where avg(price) > $20
```

is not. See "Aggregates" for information on the use of aggregate functions.

having *Without* group by

You can use a **having** clause without a **group by** clause.

If there are columns in the select list that neither have aggregate functions applied to them nor are included in the query's **group by** clause (illegal in standard SQL), the meanings of **having** and **where** are somewhat different.

In this situation, a **where** clause restricts the rows that are included in the calculation of the aggregate, but does not restrict the rows returned by the query. Conversely, a **having** clause restricts the rows returned by the query, but does not affect the calculation of the aggregate. See "**group by** and **having** Clauses" for examples.

Expressions in Search Conditions

Search conditions with expressions use the following format:

```
{where | having} [not]
   expression comparison_operator expression
```

See "Expressions" for an explanation of *expression* and the available *comparison_operator*s.

Negating Expressions

not can negate any logical expression and keywords such as **like**, **null**, **between**, **in**, and **exists**.

For example,

```
where phone not like '415%'
```

Finds all the rows in which the phone number does not begin with 415.

Using *like*

The **like** keyword indicates that the following character string (enclosed by single or double quotes) is a matching pattern. **like** is available for *char*, *varchar*, and *datetime* columns (but not to search for seconds or milliseconds). Search conditions can specify the **like** keyword in the following format:

```
{where | having} [not]
   column_name [not] like  "match_string"
```

The wildcards are:

Symbol	Meaning
%	any string of 0 or more characters
_	any single character
[ ]	any single character within the specified range ([a-f]) or set ([abcdef])
[^]	any single character not within the specified range ([^a-f]) or set ([^abcdef])

Table 3-24: Wildcards used in match strings

The wildcard and the string must be enclosed in single or double quotes. For complete information, including information on using alternative character set definitions in wildcards, see "Wildcard Characters."

The following example finds the rows for authors named Carson, Carsen, Karsen, and Karson.

```
where au_lname like "[CK]ars[eo]n"
```

You can use the keyword **like** and wildcard characters with *datetime* data as well as with *char*, *varchar* and *text*. When you use **like** with *datetime* values, SQL Server converts the dates to the standard *datetime* format, and then to *varchar*. Use the **convert** function to display seconds and milliseconds.

It is a good idea to use **like** when you search for *datetime* values, since *datetime* entries may contain a variety of date parts. For example, if you insert the value '9:20' into a column named *arrival_time*, the clause:

```
where arrival_time = '9:20'
```

would not find it because SQL Server converts the entry into "Jan 1, 1900 9:20AM." However, the clause:

```
where arrival_time like '%9:20%'
```

would find it.

Using *null*s

is [**not**] **null** is used when searching for null values (or all values except null values). It is allowed only if the column has been defined to allow null values in the **create table** statement.

An expression with a bitwise or arithmetic operator evaluates to NULL if any of the operands are null.

The general syntax is:

```
{where | having} [not] column_name is [not] null
```

For example:

```
where advance < $5000 or advance is null
```

Ranges

Specify ranges in search conditions as follows:

```
{where | having} [not]
   expression [not] between expression and expression
```

between is the range-start keyword. Use **and** for the range-end value. The range:

```
between x and y
```

is inclusive, unlike the range:

```
> x < y
```

For example:

```
where total_sales between 4095 and 12000
```

Matching Lists

The **in** keyword allows you to select values that match any one of a list of values. The expression can be a constant or a column name, and the values list can be a set of constants or, more commonly, a subquery. (See "Subqueries" for information on using **in** with a subquery.)

Enclose the list of values in parentheses. Put single or double quotes around *char*, *varchar*, and *datetime* values, and separate each value from the following one with a comma. The syntax is as follows:

```
{where | having} [not]
   expression [not] in ({value_list | subquery})
```

The following example finds the rows in which the state is one of the three in the list.

```
where state in ('CA', 'IN', 'MD')
```

Testing for Existence

The **exists** keyword is used with a subquery to test for the existence of some result from the subquery. (See "Subqueries" for more information.) The general syntax follows:

```
{where | having} [not] exists (subquery)
```

Using *any* and *all*

any is used with <, >, or = and a subquery. It returns results when any value retrieved in the subquery matches the value in the **where** or **having** clause of the outer statement. (See "Subqueries" for more information.)

all is used with **<** or **>** and a subquery. It returns results when all values retrieved in the subquery match the value in the **where** or **having** clause of the outer statement. (See "Subqueries" for more information.)

The syntax for **any** and **all** is:

```
{where | having} [not]
   expression comparison_operator {any |all} (subquery)
```

Join Operators

The syntax of join operators in search conditions is:

```
{where | having} [not]
   column_name join_operator column_name
```

where *join_operator* is a comparison operator, **=***, or ***=** (see "Joins" for more information) and *column_name* is the name of a column used in the comparison. Qualify the name of the column if there is any ambiguity.

Connecting Search Conditions

If you use more than one of the search conditions in a single statement, connect the conditions with **and** or **or**. The syntax is:

```
{where | having} [not]
   logical_expression {and | or} logical_expression
```

and joins two conditions and returns results when both of the conditions are true.

or joins two conditions and returns results when either of the conditions is true.

When more than one logical operator is used in a statement, **and** operators are normally evaluated before **or** operators. You can change the order of execution with parentheses. For example:

```
where (type = "business" or type = "psychology")
and
advance > $5500
```

Standards and Compliance

Standard	Compliance Level	Comments
SQL89	Compliant	The following are Transact-SQL extensions: • Modulo operator % • Negative comparison operators !>, !<, and != • Bitwise operators ~, ^, |, and & • Join operators *= and =* • Wildcards [] and - • Not operator ^
SQL92	Entry level compliant	

See Also

Commands	**delete**, **execute**, **insert**, **select**, **update**
Functions	Aggregate Functions
Topics	Expressions, Joins, Subqueries, Wildcard Characters
System procedures	**sp_helpjoins**

Subqueries

Function

Subqueries "nest" a **select** statement inside a **select**, **insert**, **update**, or **delete** statement, another subquery, or anywhere an expression is allowed (if it returns a single value). A subquery is always enclosed in parentheses.

How to Define Subqueries

You can use a subquery almost anywhere an expression can be used if it returns a single value. You must always enclose subqueries in parentheses. For example:

```
select distinct pub_name
   from publishers
   where "business" in
       (select type from titles
       where pub_id = publishers.pub_id)
```

Subqueries **cannot** be used in an **order by** list. *text* and *image* datatypes are not allowed in subqueries. Be sure to qualify column names with table or view names if there is ambiguity about the table or view to which they belong.

A subquery has this restricted **select** syntax:

```
(select [all | distinct] subquery_select_list
   [from [[database.]owner.]{table_name | view_name}
       [holdlock]
   [, [[database.]owner.]{table_name | view_name}
       [holdlock]]...]
   [where search_conditions]
   [group by aggregate_free_expression
       [, aggregate_free_expression]...]
   [having search_conditions])
```

The *subquery_select_list* is a restricted *select_list* for use with subqueries introduced with **in** or a comparison operator. It consists of one expression. If a column name is used in the **where** clause of the outer statement, a column name in the *subquery_select_list* must be join compatible with it.

The *select_list* must consist of only one column name, except in the **exists** subquery, in which case the asterisk (*) is usually used in place of the single column name. Do not specify more than one column name.

See **select** for more information on the other options. See "**where** Clause" for more information on search conditions and **where** clause syntax.

Many SQL statements that include subqueries can be alternatively formulated as joins. See "Joins" for more information on joining.

Using Subqueries with Comparison Operators

The following is the syntax for using subqueries with comparison operators in a **select**, **insert**, **update**, or **delete** statement:

expression comparison_operator **[any | all] (s*ubquery*)**

An *expression* is a column name, a constant, a function, any combination of column names, constants, and functions connected by arithmetic or bitwise operators, or a subquery. (This and other formats for subqueries are considered expressions in and of themselves, and can be substituted wherever the word "expression" appears as long as the subquery returns a single value.)

The arithmetic operators are:

Symbol	Operation
+	addition
-	subtraction
/	division
*	multiplication
%	modulo

Table 3-25: Arithmetic operators

Addition, subtraction, division, and multiplication can be used on *int*, *smallint*, *tinyint*, *float*, and *money* columns. The modulo operator can be used with all integer datatypes, but cannot be used with *money* or *float* columns. A modulo is the integer remainder after a division involving two integers. For example, 21 % 9 = 3 because 21 divided by 9 equals 2 with a remainder of 3.

The bitwise operators are:

Symbol	Meaning
&	bitwise and (two operands)
|	bitwise or (two operands)
^	bitwise exclusive or (two operands)
~	bitwise not (one operand)

Table 3-26: Bitwise operators

The bitwise operators can be used only on *int*, *smallint*, and *tinyint* columns. All the bitwise operators translate the integer arguments into binary representation before evaluating them. (See "Expressions" for more information.)

The *comparison_operator* is one of the following items:

Symbol	Meaning
=	equal to
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
!=	not equal to
<>	not equal to
!>	not greater than
!<	not less than

Table 3-27: Comparison operators

In comparing *char*, *nchar*, *varchar*, and *nvarchar* data, **<** means closer to the beginning of the alphabet and **>** means closer to the end of the alphabet. Case and special character evaluations depend on the collating sequence of the machine on which SQL Server is located.

For example, lowercase letters may be greater than uppercase letters, and uppercase letters may be grea

ter than numbers. In comparing dates, **<** means earlier and **>** means later. Trailing blanks are ignored for comparison purposes. For example, "Dirk" is the same as "Dirk ".

Put quotes around all character and *datetime* data used with a comparison operator:

```
= "Bennet"
> "94609"
```

subquery is a query with the restricted **select** syntax shown at the beginning of this section. It cannot include **order by** or **compute** clauses or the keyword **into**. The subquery select list must consist of only one column name, except in the **exists** subquery, in which case the asterisk (*) is almost always used for the subquery select list.

A subquery can contain another subquery. A subquery is legal in any Transact-SQL syntax that allows an expression, as long as it returns a single value.

The **all** keyword is used with **>**, **<**, or **=** and a subquery. **> all** means that the value being compared must be greater than **all** the values in the list (that is, greater than the maximum value). **< all** means that the value being compared must be less than **all** the values in the list (that is, less than the minimum value). **= all** means that the value being compared must be equal to **all** the values in the list.

The following example finds the books that commanded an advance greater than the largest advance paid by Algodata Infosystems:

```
select title from titles
   where advance > all
     (select advance from titles, publishers
      where titles.pub_id = publishers.pub_id
      and pub_name = "Algodata Infosystems")
```

The **any** keyword is used with **>**, **<**, or **=** and a subquery. **> any** is true (and the subquery returns results) if the value in the outer query is greater than any value in the list (that is, greater than the minimum value in the list). **< any** is true (and the subquery returns results) if the value in the outer query is less than any value in the list (that is, less than the maximum value in the list).

The following example finds the authors that live in the same city as some publisher:

```
select au_lname, au_fname from authors
   where city = any
      (select city
       from publishers)
```

Using Subqueries with *in* and *not in*

The following is the syntax for using subqueries with the **in** and **not in** keywords in a **select**, **insert**, **update**, or **delete** statement:

expression **[not] in (*subquery*)**

The **in** keyword returns results when any value in the subquery matches the value in the **where** clause of the outer statement. (**in** is equivalent to **= any**.) You can also use **in** with a list of values enclosed in parentheses. See "**where** Clause" for details.

The following example uses **in** to find all the publishers that publish business books:

```
select distinct pub_name from publishers
   where pub_id in
     (select pub_id from titles
     where type = "business")
```

The next example uses **in** to find the names of authors who have participated in writing at least one popular computing book (note that it nests one subquery within another):

```
select au_lname, au_fname from authors
   where au_id in
     (select au_id from titleauthor
     where title_id in
       (select title_id from titles
       where type = "popular_comp"))
```

Subqueries introduced with **not in** can take a long time to run, since they must examine the data in each row of the inner query. Indexes cannot return information about what is not in a table.

In contrast, subqueries introduced with **in** can use indexes if they exist on the column, and can return results as soon as they find a single value that matches. For complex queries, frequently run queries, or large tables, you may want to find another way to express the query.

Using Subqueries with *exists*

The following is the syntax for using subqueries with the **exists** keyword in a **select**, **insert**, **update**, or **delete** statement:

[not] exists (*subquery*)

The **exists** keyword is used with a subquery to test for the existence of some result from the subquery. **exists** is not preceded by an expression. The subquery that follows the keyword **exists** is enclosed in parentheses. If the **not** option is used, the **where** clause is satisfied if no rows are returned by the subquery. Since **exists** tests only for the existence of a condition, the asterisk (*) is almost always used as the select list.

The following example also finds the names of publishers who have published business books using **exists**:

```
select distinct pub_name from publishers
   where exists
     (select * from titles
      where pub_id = publishers.pub_id
      and type = "business")
```

The syntax for **exists** is somewhat different from the syntax for the other keywords; it does not take an expression between **where** and itself. The following two queries, which are semantically equivalent, illustrate the difference:

```
select distinct pub_name from publishers
   where exists
   (select * from titles
   where pub_id = publishers.pub_id
   and type = "business")

select distinct pub_name from publishers
   where pub_id in
   (select pub_id from titles
   where type = "business")
```

Correlated Subqueries

A correlated (or repeating) subquery depends on the outer query for its values. It is executed repeatedly, once for each row that might be selected by the outer query. For example, the subquery in the following statement cannot be evaluated independently of the outer query. The value for *authors.au_id* changes as the outer query examines each row in *authors*:

```
select distinct au_lname, au_fname from authors
   where 100 in
       (select royaltyper from titleauthor
       where titleauthor.au_id = authors.au_id)
```

A correlated subquery can also be used in the **having** clause of an outer query. This example finds the types of books for which the maximum advance is more than twice the average for the group:

```
select t1.type from titles t1
   group by t1.type
   having max(t1.advance) >= all
       (select 2 * avg(t2.advance) from titles t2
       where t1.type = t1.type)
```

You **cannot** use correlated subqueries in the **select** clause of an updatable cursor defined by **declare cursor.**

See Also

Commands	**delete**, **execute**, **group by** and **having** Clauses, **insert**, **select**, **update**, **where** Clause
Topics	Cursors, Expressions, Joins

Temporary Tables

Two Types of Temporary Tables

Temporary tables are created in the *tempdb* database, not the current database. To create a temporary table, you must have **create table** permission in *tempdb*. **create table** permission defaults to *public*.

There are two kinds of temporary tables:

- Tables that are accessible only by the current session and exist only for the duration of the current session.
- Tables that are accessible by any user on any session and exist either until specifically dropped or until SQL Server reboots.

Tables Accessible Only by the Current Session

A non-sharable temporary table exists until the current session ends, or until its owner drops it using **drop table**. You create a temporary table which cannot be shared by specifying only the table's name in the **create table** statement. The table name must begin with a pound (#) sign. For example:

```
create table #my_temptbl
```

SQL Server Appends a Suffix to the Table Name

To ensure that the table name is unique for the current session, SQL Server:

- Truncates long temporary table names to 13 characters (including the pound sign)
- Pads short names to 13 characters, using underscores (_)
- Appends a 17-digit numeric suffix that is unique for a SQL Server session

The following example shows a table created as *#temptable* and stored as *#temptable___00000050010721973*:

```
use pubs2
go
create table #temptable (task char(30))
go
use tempdb
go
select name from sysobjects where name like
    '#temptable%'
go
```

```
name
 ------------------------------
#temptable___00000050010721973
(1 row affected)
```

Restrictions on Temporary Tables

Temporary tables with names that begin with # are subject to the following restrictions:

- You cannot create views on these tables.
- You cannot associate triggers with these tables.

These restrictions do not apply to sharable temporary tables created in *tempdb*.

Tables That Can Be Shared Among Sessions

You create these sharable temporary tables by specifying *tempdb* as part of the table's name in the **create table** statement, or by creating them directly in *tempdb*. The following examples both create a sharable temporary table, *my_temptbl*, in *tempdb*:

```
1. create table tempdb..my_temptbl

2. use tempdb
   create table my_temptbl
```

SQL Server does not change the names of temporary tables created this way. The table exists either until you reboot SQL Server, or until its owner drops it using **drop table**. Temporary tables are not recoverable.

Using Rules, Defaults, and Indexes

You can associate rules, defaults and indexes with temporary tables. Indexes created on a temporary table disappear when the temporary table disappears.

Manipulating Temporary Tables in Stored Procedures

Stored procedures can reference temporary tables that are created during the current session. Within a stored procedure, you cannot create a temporary table, drop it, and then create a new temporary table with the same name.

Temporary Tables with Names Beginning with #

Temporary tables with names beginning with # that are created within stored procedures disappear when the procedure exits. A single procedure can:

- Create a temporary table
- Insert data into it
- Run queries on it
- Call other procedures that reference the temporary table

Since the temporary table must exist in order to create procedures that reference it, here are the steps to follow:

1. Create the temporary table you need with a **create table** statement.
2. Create the procedures that access the temporary table (but not the one that creates it).
3. Drop the temporary table.
4. Create the procedure that creates the table and calls the procedures created in Step 2.

Tables with Names Beginning with tempdb..

You can also create temporary tables without the # prefix, using **create table** ***tempdb..tablename*** from inside a stored procedure. These tables do not disappear when the procedure completes, so they can be referenced by independent procedures. Follow the steps listed above to create these tables.

Using System Procedures on Temporary Tables

System procedures such as **sp_help** only work on temporary tables if you invoke them from *tempdb*.

Using User-Defined Datatypes in Temporary Tables

User-defined datatypes cannot be used in temporary tables unless the datatypes exist in *tempdb*; that is, unless they have been explicitly created in *tempdb* since the last time SQL Server was rebooted.

Selecting into Temporary Result Tables

You do not have to set the **select into/bulkcopy** option on to **select into** a temporary table.

Standards and Compliance

Temporary tables are a Transact-SQL extension.

See Also

Commands	**create table, drop table**

text and image Datatypes

Function

text columns are variable-length columns that can hold up to 2,147,483,647 (2^{31} - 1) bytes of printable characters.

image columns are variable-length columns that can hold up to 2,147,483,647 (2^{31} - 1) bytes of hexadecimal-like data.

Defining a *text* or *image* Column

You define a text or image column as you would any other column, with a **create table** or **alter table** statement. *text* and *image* datatype definitions do not include lengths. They do permit null values. The column definition takes the form:

column_name **{text | image} [null]**

For example, to create the table *texttest* with a *text* column, *blurb*, that permits null values:

```
create table texttest
(title_id varchar(6), blurb text null, pub_id char(4))
```

And to create the *imagetest* table with an *image* column:

```
create table imagetest
(image_id varchar(6), imagecol image null,
graphic_id char(4))
```

Data Is Stored on a Linked List of Pages

SQL Server stores text and image data in a linked list of data pages (2K for most systems; 4K for Stratus) separate from the rest of the table. Each *text* or *image* page stores a maximum of 1800 bytes of data. For a table with more than one *text* or *image* column, all the data is stored in the same chain of pages.

Initializing *text* and *image* Columns

text and *image* columns are not initialized until you update them or insert a non-null value. Initialization allocates at least one data page for each non-null *text* or *image* data value. It also creates a pointer in the table to the location of the *text* or *image* data. To save storage space, **insert** null values until you use the *text* or *image* column.

For example, the following statement initializes the *blurb* column by inserting a non-null value. The column now has a valid text pointer,

and the first 2K data page has been allocated. Note that *text* values require quotation marks in an **insert** statement:

```
insert texttest values
("BU7832", "Straight Talk About Computers is an
annotated analysis of what computers can do for
you: a no-hype guide for the critical user.",
"1389")
```

The following statement initializes an image column. Note that *image* values must be preceded by 0x in an **insert** statement:

```
insert imagetest values
("94732", 0x0000008300000000000100000000013c,
"1389")
```

How *image* Data Is Stored

image values less than 255 bytes that have an odd number of bytes are padded with a leading zero (an insert of "0xaaabb" becomes "0x0aaabb"). It is an error to insert *image* values greater than 255 bytes that have an odd number of bytes.

Saving Space by Allowing Nulls

To save storage space for empty *text* or *image* columns, define them to permit null values. Inserting a null value does not initialize a *text* or *image* column, and therefore does not create a text pointer or allocate 2K bytes of storage. For example, the following statement inserts values into the *title_id* and *pub_id* columns, but does not initialize the *blurb* text column:

```
insert texttest
(title_id, pub_id) values ("BU7832", "1389")
```

Getting Information from *sysindexes*

Each table with *text* or *image* columns has an additional row in *sysindexes* that provides information about these columns. The *name* column in *sysindexes* uses the form "t*tablename*"; the *indid* is always 255. These columns provide information about text storage:

Column	Description
ioampg	Pointer to the allocation page for the text page chain
first	Pointer to the first page of text data
root	Pointer to the last page
segment	Number of the segment where the object resides

Table 3-28: Storage of text and image data

You can query the *sysindexes* table for information about these columns. For example, the following query reports the number of data pages used by the *blurbs* table in the *pubs2* database:

```
select name, data_pgs(object_id("blurbs"), ioampg)
from sysindexes
where name = "tblurbs"

name
------------------------------ -----------
 tblurbs                        7
```

Using *readtext* and *writetext*

The DB-Library functions **dbwritetext** and **dbmoretext** are the most efficient way to enter large *text* and *image* values. To use **writetext** to enter *text* data or **readtext** to read it, the *text* column **must** be initialized first. See "**readtext**" and "**writetext**" for more details.

Using **update** to replace existing *text* and *image* data with NULL, reclaims all of the allocated data pages except the first page, which remains available for future use of **writetext**. To deallocate all of the storage for the row, use **delete** to remove the entire row.

Determining How Much Space a Column Uses

The system procedure **sp_spaceused** provides information about the space used for text data as *index_size*:

```
sp_spaceused blurbs
name            rowtotal  reserved  data    index_size unused
--------------- --------  --------- ------- ---------- ------
blurbs          6         32 KB     2 KB    14 KB      16 KB
```

Restrictions on *text* and *image* Columns

text and *image* columns cannot be used:

- As parameters to stored procedures or as values passed to these parameters
- As local variables
- In **order by, compute**, and **group by** clauses
- In an index
- In subqueries
- In a **where** clause, except with the keyword **like**
- In joins

Putting Additional Pages on Another Device

You can place subsequent *text* and *image* data pages on a different logical device with **sp_placeobject**.

Selecting *text* and *image* Data

select statements return *text* and *image* data up to the limit specified in the global variable @@*textsize*. The **set textsize** command is used to change this limit. The initial value of @@*textsize* is 32K; the maximum value for @@*textsize* is 2^{31} - 1 (that is, 2,147,483,647).

Converting the *text* and *image* Datatypes

You can explicitly convert *text* values to *char* or *varchar* and *image* values to *binary* or *varbinary* with the **convert** function, but you are limited to the maximum length of the character and binary datatypes, 255 bytes. If you do not specify the length, the converted value has a default length of 30 bytes. Implicit conversion is not supported.

Pattern Matching in *text* Data

Use the **patindex** function to search for the starting position of the first occurrence of a specified pattern in a *text*, *varchar*, or *char* column. The % wildcard character must precede and follow the pattern (except when searching for first or last characters).

You can also use the **like** keyword to search for a particular pattern. The following example selects each *text* data value from the *blurb* column of the *texttest* table that contains the pattern, "Straight Talk%".

```
select blurb from texttest
where blurb like "Straight Talk%"
```

Duplicate Rows Are Prohibited

Because the pointer to the *text* or *image* data uniquely identifies each row, a table that contains *text* or *image* data cannot contain duplicate rows unless the pointer has not been initialized: that is, unless all *text* and *image* data is NULL.

See Also

Commands	**insert**, **readtext**, **set**, **update**, **writetext**
Functions	Datatype Conversion Functions, Text and Image Functions
Topics	Datatypes

Transactions

Function

User-defined transactions provide a mechanism for grouping statements so that they are treated as a unit: either all statements in the group are executed, or no statements in the group are executed.

About Transactions

To explicitly define the beginning of a transaction:

```
begin {transaction | tran | work} [transaction_name]
```

transaction_name is the name assigned to the transaction. It must conform to the rules for identifiers. Use transaction names only on the outermost pair of nested **begin** / **commit** or **begin** / **rollback** statements.

To commit a transaction:

```
commit {transaction | tran | work}
    [transaction_name]
```

To roll a transaction back to a savepoint, or to the beginning of a transaction:

```
rollback {transaction | tran | work}
    [transaction_name | savepoint_name]
```

savepoint_name is the name assigned to the savepoint. It must conform to the rules for identifiers.

To mark a savepoint within a transaction:

```
save {transaction | tran} savepoint_name
```

begin transaction marks the beginning of a transaction block. All subsequent statements, up to a **rollback** or a matching **commit**, are treated as a unit: they are all rolled back (if unsuccessful) or committed (if successful).

To get full ANSI-compliant transactions, you must set the **chained** and **transaction isolation level 3** options at the beginning of every application which changes the mode and isolation level for subsequent transactions. If your application uses cursors, you must also set the **close on endtran** option. Each of these options is described later.

The global variable *@@transtate* keeps track of the current state of a transaction. SQL Server determines what state to return by keeping

track of any transaction changes after a statement executes. *@@transtate* may contain the following values:

Value	Meaning
0	Transaction in progress: an explicit or implicit transaction is in effect; the previous statement executed successfully.
1	Transaction succeeded: the transaction completed and committed its changes.
2	Statement aborted: the previous statement was aborted; no effect on the transaction.
3	Transaction aborted: the transaction aborted and rolled back any changes.

Table 3-29: @@transtate values

SQL Server does not clear *@@transtate* after every statement. You can use *@@transtate* after a statement (such as an **insert**) to determine whether it was successful or aborted and to determine its effect on the transaction.

Example of Using a Transaction

A user sets out to change the royalty split for the two authors of *The Gourmet Microwave*. Since the database would be inconsistent between the two updates, they must be grouped into a user-defined transaction. For example:

```
begin transaction royalty_change

update titleauthor
set royaltyper = 65
from titleauthor, titles
where royaltyper = 75
and titleauthor.title_id = titles.title_id
and title = "The Gourmet Microwave"

update titleauthor
set royaltyper = 35
from titleauthor, titles
where royaltyper = 25
and titleauthor.title_id = titles.title_id
and title = "The Gourmet Microwave"

save transaction percentchanged
```

```
/* After updating the royaltyper entries for
** the two authors, insert the savepoint
** percentchanged, then determine how a 10%
** increase in the book's price would affect
** the authors' royalty earnings. */

update titles
set price = price * 1.1
where title = "The Gourmet Microwave"

select (price * total_sales) * royaltyper
from titles, titleauthor
where title = "The Gourmet Microwave"
and titles.title_id = titleauthor.title_id

/* The transaction is rolled back to the savepoint
** with the rollback transaction command. */

rollback transaction percentchanged

commit transaction
```

Nesting Transactions

You can nest **begin transaction** / **commit** statements. The outermost pair actually create and commit the transaction; inner pairs just keep track of the nesting level. The transaction is not committed until the **commit** that matches the outermost **begin transaction** (explicit or implicit) is issued. Normally, this transaction "nesting" occurs as stored procedures or triggers which contain **begin** / **commit** pairs call each other.

The global variable *@@trancount* keeps track of the nesting levels of transactions. A *@@trancount* of 0 indicates no current transaction. An initial explicit or implicit **begin transaction** sets *@@trancount* to 1. Each subsequent explicit **begin transaction** increments *@@trancount*. **commit** decrements *@@trancount*; nested transactions are not committed until *@@trancount* reaches 0. For example:

```
begin tran
    statements              Group A, @@trancount = 1
    begin tran
        statements          Group B, @@trancount = 2
        begin tran
          statements        Group C, @@trancount = 3
        commit tran
    commit tran
commit tran                 All statement groups committed here
```

Considerations When Using Transactions

You should consider the following issues when dealing with transactions in your applications:

- A **rollback** statement, without a transaction or savepoint name, always rolls back statements to the outermost **begin transaction** (explicit or implicit) statement, and cancels the transaction. If there is no current transaction when you issue **rollback**, the statement has no effect.

  In triggers or stored procedures, **rollback** statements, without transaction or savepoint names, roll back all statements to the outermost **begin transaction** (explicit or implicit).

- **rollback** does not produce any messages to the user. If warnings are needed, use **raiserror** or **print** statements.
- Grouping a large number of Transact-SQL commands into one long-running transaction may affect recovery time. If SQL Server fails during a long transaction, recovery time increases, since SQL Server must first undo the entire transaction.
- You can refer to as many as 8 databases within a transaction. However, SQL Server may refer to internal databases to process a transaction, so the number of databases you can actually refer to may be lower. SQL Server displays an error message if you exceed the open databases limit.
- A remote procedure call (RPC) is executed independently from any transaction in which it is included. In a standard transaction (that is, not using Open Client DB-Library / C two-phase commit), commands executed via an RPC by a remote server are not rolled back with **rollback**, and do not depend on **commit** to be executed.
- Transactions cannot span more than one connection between a client application and a server. For example, an Open Client DB-Library / C application cannot group SQL statements in a transaction across multiple open DBPROCESSes.

Transactions and Data Definition Commands

You can use certain data definition language commands in transactions if **sp_dboption**'s **ddl in tran** option is set to true. If **ddl in tran** is true in a particular database, commands such as **create table**, **grant**, and **alter table** are allowed inside transactions in that database. If **ddl in tran** is true in the *model* database, the commands are allowed inside transactions in all databases created after **ddl in tran** was set in *model*.

◆ ***WARNING!***

The only scenario in which using data definition language commands inside transactions is justified is in create schema. Data definition language commands hold locks on system tables such as *sysobjects*. If you use data definition language commands inside transactions, keep the transactions short.

In particular, avoid using any data definition language commands on *tempdb* within transactions, lest your system grind to a halt. Always leave ddl in tran set to false in *tempdb*.

To set **ddl in tran** to true for a database (such as *mydb*), move to the *master* database and type:

```
sp_dboption mydb, "ddl in tran", true
```

You must then execute the **checkpoint** command in that database.

The following commands can be used inside a user-defined transaction only if the **ddl in tran** option to **sp_dboption** is set to true:

alter table	create default	drop default	grant
	create index	drop index	revoke
	create procedure	drop procedure	
	create rule	drop rule	
	create schema	drop table	
	create table	drop trigger	
	create trigger	drop view	
	create view		

Table 3-30: DDL commands allowed in transactions

The following commands cannot be used inside a user-defined transaction under any circumstances:

alter database	dump transaction	reconfigure
create database	drop database	select into
disk init	load transaction	update statistics
dump database	load database	truncate table

Table 3-31: DDL commands not allowed in transactions

In addition, the **sp_helpdb**, **sp_helpdevice**, **sp_helpindex**, **sp_helpjoins**, **sp_helpserver**, **sp_lookup**, **sp_spaceused**, and **sp_syntax** system procedures cannot be used inside user-defined transactions because they create temporary tables.

System procedures that change the *master* database cannot be used inside user-defined transactions.

You can check the current setting of **ddl in tran** with **sp_helpdb**.

Using Savepoints in Transactions

save transaction *transaction_name* provides a mechanism for selectively rolling back portions of a batch. You can commit only certain portions of a batch by rolling back the undesired portion to a savepoint before committing the entire batch. For example:

```
begin tran
    statements                    Group A
    save tran mytran
        statements                Group B
    rollback tran mytran          Rolls back group B
    statements                    Group C
commit tran                       Commits groups A and C
```

You can also use **save transaction** to create transactions within stored procedures or triggers in such a way that they can be rolled back without affecting batches or other procedures. For example:

```
create proc myproc as
begin tran
save tran mytran
statements
if ...
    begin
        rollback tran mytran
    /*
    ** Rolls back to savepoint.
    */
        commit tran
    /*
    ** This commit needed; rollback to a savepoint
    ** does not cancel a transaction.
    */
     end
```

```
else
    commit tran
    /*
    ** Matches begin tran; either commits
    ** transaction (if not nested) or
    ** decrements nesting level.
    */
```

Unless you are rolling back to a savepoint, use transaction names only on the outermost pair of **begin transaction** / **commit** or **begin transaction** / **rollback** statements.

◆ ***WARNING!***

Transaction names are ignored, or can cause errors, when used in nested transaction statements. If you are using transactions in stored procedures or triggers that could be called from within other transactions, do not use transaction names.

Choosing a Transaction Mode

SQL Server supports two transaction modes:

- The default mode, called **unchained** or Transact-SQL mode, requires explicit **begin transaction** statements paired with **commit** or **rollback** statements to complete the transaction.
- The ANSI-compatible mode, called **chained** mode, implicitly begins a transaction before any data retrieval or modification statement. These statements include: **delete**, **insert**, **open**, **fetch**, **select**, and **update**. You must still explicitly end the transaction with **commit** or **rollback**.

You can set either mode using the **chained** option of the **set** command. For example:

```
set chained on
```

However, do not mix these transaction modes in your applications. The behavior of batches, stored procedures, and triggers can vary depending on the mode.

For example, the following group of statements produce different results depending on which mode you use:

```
insert into publishers
    values ('9999', null, null, null)
begin transaction
delete from publishers where pub_id = '9999'
rollback transaction
```

In unchained mode, the **rollback** affects only the **delete** statement, so *publishers* still contains the inserted row. In chained mode, the **insert** statement implicitly begins a transaction, and the rollback affects all statements up to the beginning of that transaction, including the **insert.**

All application programs and ad-hoc user queries should know their current transaction mode. Which transaction mode you use depends on whether or not a particular query or application requires ANSI compliance. Applications that will use chained transactions (for example, the Embedded-SQL precompiler) should set chained mode at the beginning of each session.

In chained mode, a data retrieval or modification statement begins a transaction whether or not it executes successfully. Even a **select** which does not access a table begins a transaction. However, a **fetch** begins a transaction only when an application has not set **close on endtran**, which allows cursors to remain open across transactions.

The global variable *@@tranchained* contains the current transaction mode of the Transact-SQL program. *@@tranchained* returns 0 for unchained and 1 for chained.

You cannot execute the **set chained** command within a transaction. If you do, SQL Server issues a level 16 (informational) error message, and the current transaction continues unchanged.

Choosing a Transaction Isolation Level

The SQL92 standard defines three levels of isolation for transactions. Each isolation level specifies the kinds of actions which are not permitted while concurrent transactions execute. Higher levels include the restrictions imposed by the lower levels:

- Level 1 prevents **dirty reads**. "Dirty reads" occur when one transaction modifies a row, and then a second transaction reads that row before the first transaction commits the change. If the first transaction rolls back the change, the information read by the second transaction becomes invalid. Transact-SQL never allows this to happen.

- Level 2 prevents **non-repeatable reads**. "Non-repeatable reads" occur when one transaction reads a row and then a second transaction modifies that row. If the second transaction commits its change, subsequent reads by the first transaction yield different results than the original read. Transact-SQL supports this level of isolation through the **holdlock** option of the **select** statement. **holdlock** applies a read-lock on the data until the transaction ends.
- Level 3 prevents **phantoms**. "Phantoms" occur when one transaction reads a set of rows that satisfy a search condition, and then a second transaction modifies the data (through an **insert**, **delete**, **update**, and so on). If the first transaction repeats the read with the same search conditions, it obtains a different set of rows. Transact-SQL also supports this level through **holdlock**.

The SQL92 standard requires that level 3 be the default isolation between transactions. This prevents "dirty reads," "non-repeatable reads," and "phantoms." To enforce this default level of isolation, Transact-SQL provides the **transaction isolation level 3** option of the **set** statement. This instructs SQL Server to automatically apply a **holdlock** to all **select** operations in a transaction.

By default, SQL Server's transaction isolation level is 1. You cannot change an isolation level within a transaction. Such attempts cause SQL Server to return a warning message; the current transaction continues unchanged. Applications that should use **transaction isolation level 3** (for example, the Embedded-SQL precompiler) should set that isolation level at the beginning of each session.

Setting **transaction isolation level 3** causes SQL Server to hold any read-locks for the duration of the transaction. If you also use the chained mode, that isolation level remains in effect for any data retrieval or modification statement that implicitly begins a transaction. This can lead to performance penalties for some applications since more locks may be held for longer periods of time.

The global variable *@@isolation* determines the current isolation level of the Transact-SQL program. *@@isolation* returns the value of the active level (1 or 3). For example:

```
select @@isolation

----------
         1

(1 row affected)
```

Using Transactions in Stored Procedures and Triggers

A **rollback** statement in a stored procedure or trigger that rolls back past more than one explicit or implicit **begin transaction** statement produces an informational message indicating the number of nesting levels affected. This message does not affect subsequent processing.

The illustration that follows demonstrates a sequence of events that might occur due to nested transaction statements in triggers and procedures.

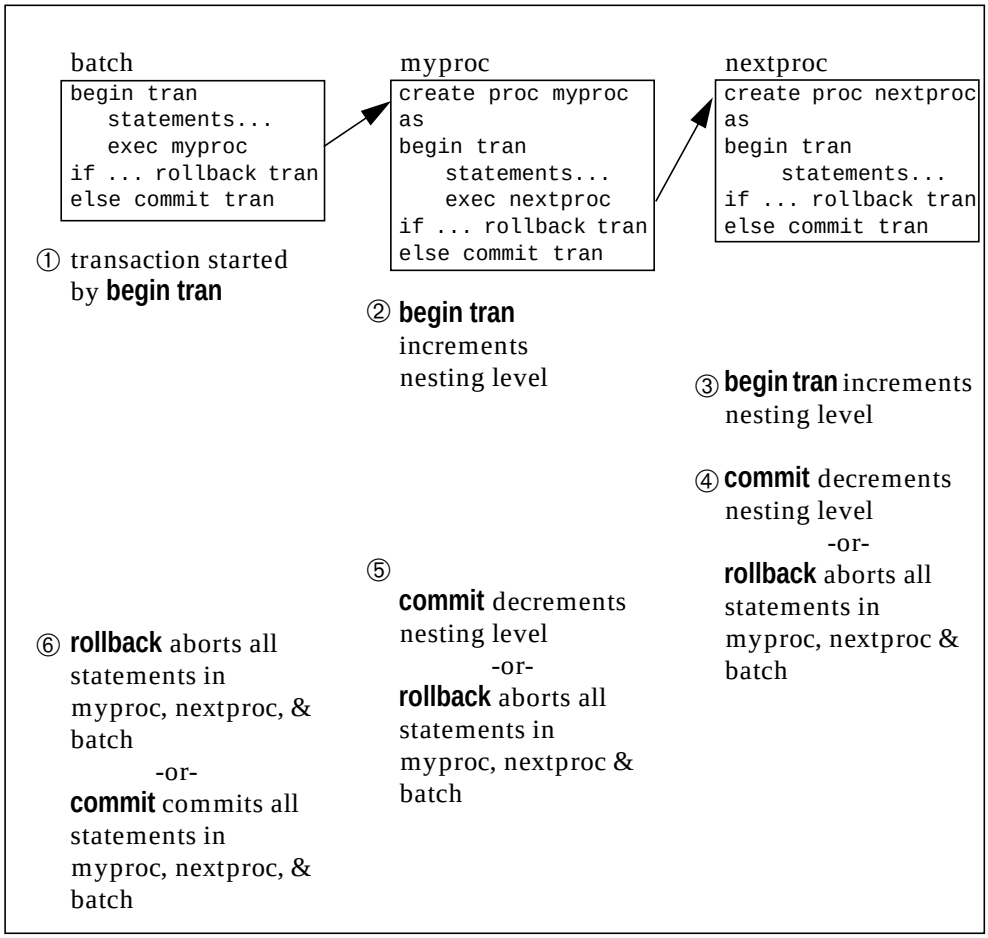


Figure 3-1: Nesting transaction statements

rollback statements in stored procedures do *not* affect subsequent statements in the procedure or batch that called the procedure. Subsequent statements in the stored procedure and / or batch are executed.

➤ ***Note***

rollback statements in trigger: 1) roll back the transaction, 2) complete subsequent statements in the trigger, and 3) abort the batch so that subsequent statements in the batch are **not** executed.

In the following example, the procedure *myproc* includes a **rollback** statement. The **update** and **insert** statements are rolled back and the transaction is aborted. The **delete** statement is executed, and cannot be rolled back.

```
begin tran
update mytab ...
insert abc values ...
execute myproc
delete mytab ...
```

However, in the following example, if there is an **insert** trigger on *xyz* that includes a **rollback**, the entire transaction is rolled back, and the **delete** is not performed:

```
begin tran
update mytab ...
insert xyz values ...
delete mytab ...
```

The following table summarizes how **rollback** affects SQL Server processing in several different contexts (such as within a transaction, stored procedure, or trigger):

Context	Rollback Effect
Transaction only	All data modifications since the start of the transaction are rolled back. If a transaction spans multiple batches, rollback affects all those batches. Any commands after the rollback are executed.
Stored procedure only	None.

Table 3-32: How rollbacks affect processing

Context	Rollback Effect
Stored procedure in a transaction	All data modification since the start of the transaction are rolled back. If a transaction spans multiple batches, rollback affects all those batches. Any commands after the rollback are executed. Stored procedure produces error message 266: "Transaction count after EXECUTE indicates that a COMMIT or ROLLBACK TRAN is missing."
Trigger only	Trigger completes, but trigger effects are rolled back. Any remaining commands in the batch are not executed. Processing resumes at the next batch.
Trigger in a transaction	Trigger completes, but trigger effects are rolled back. All data modification since the start of the transaction are rolled back. If a transaction spans multiple batches, rollback affects all those batches. Any remaining commands in the batch are not executed. Processing resumes at the next batch.
Nested trigger	Inner trigger completes, but all trigger effects are rolled back. Any remaining commands in the batch are not executed. Processing resumes at the next batch.
Nested trigger in a transaction	Inner trigger completes, but all trigger effects are rolled back. All data modification since the start of the transaction are rolled back. If a transaction spans multiple batches, rollback affects all those batches. Any remaining commands in the batch are not executed. Processing resumes at the next batch.

Table 3-32: How rollbacks affect processing (continued)

In stored procedures and triggers, the number of **begin transaction** statements must match the number of **commit** statements. A procedure or trigger that contains unpaired **begin** / **commit** statements produces a warning message when it is executed. This also applies to stored procedures that use chained mode: the first statement that implicitly begins a transaction must have a matching **commit.**

Stored Procedures and Transaction Modes

Stored procedures written to use unchained transactions may be incompatible with other chained transactions, and vice-versa. As a

rule, an application using one mode (chained or unchained) should call stored procedures written to use that same mode. This rule also applies for stored procedures using transaction isolation levels 1 or 3. The exceptions are SYBASE system stored procedures, which can be invoked by sessions using either chained or unchained transaction modes.

SQL Server allows transactions using either isolation level to invoke system stored procedures. However, you should avoid executing system procedures with isolation level 3 since they lock the system tables for longer periods. If no transaction is active when you execute system stored procedures,SQL Server turns off chained mode and sets **transaction isolation level 1** for the duration of the procedure. Before returning, these stored procedures reset the session's chained mode and isolation level to their original settings.

To avoid problems associated with transactions that use one mode invoking stored procedures that use the other mode, SQL Server tags all procedures with the transaction mode of the session in which they are created. A stored procedure tagged as "chained" is executable only in sessions using chained transactions, and one tagged as "unchained" is executable only in sessions not using chained transactions.

If you attempt otherwise, SQL Server displays a level 16 (informational) message and the current transaction, if any, continues unchanged. A third tag, "anymode", is available to indicate stored procedures that can run under either transaction mode. You can change the tag value associated with stored procedures using the **sp_procxmode** system stored procedure.

Triggers are executable under any transaction mode. Since they are always called as part of a data modification statement, they are part of a chained transaction if a session uses the chained mode, or they maintain their current transaction mode.

Cursors and Transactions

A cursor's state (open or closed) does not change when a transaction ends through a commit or abort, unless the **close on endtran** option or the chained mode is set. The **set close on endtran** statement enforces ANSI-standard behavior: it associates an open cursor with its active transaction. Committing or rolling back that transaction automatically closes any open cursors associated with it. Opening a cursor when the chained mode is set starts a transaction. SQL Server closes that cursor when the chained transaction is closed or rolled back.

For example, the following statements:

```
open cursor test
commit tran
open cursor test
```

result in a Transact-SQL error because the cursor's open state does not change when its transaction ends. However, if you set **close on endtran** or use the chained mode, the cursor's state changes from open to closed after the **commit tran**. This allows the cursor to be reopened.

➤ ***Note***

Since client applications buffer rows returned through cursors and allow users to scroll within those buffers, those client applications should not scroll backwards after a transaction aborts. The rows in a client cache may become invalid because of a transaction rollback (unknown to the client) enforced by the **close on endtran** option or the chained mode.

Any exclusive locks acquired by a cursor in a transaction are held until the end of that transaction. This also applies to shared locks when using the **holdlock** keyword or the **set isolation level 3** option. However, if you do not set the **close on endtran** option, the cursor remains open past the end of the transaction, and its current page lock remains in effect. It could also continue to acquire locks as it fetches additional rows.

The following rules define the behavior of updates through a cursor with regard to transactions:

- If an update occurs within an explicit transaction (that is, client specified), the update is considered part of the transaction. If the transaction commits, any updates included with the transaction also commit. If the transaction aborts, any updates included with the transaction are rolled back. Updates through the same cursor that occurred outside the aborted transaction are not affected.
- If updates through a cursor occur within an explicit transaction, SQL Server does not commit them when the cursor is closed. It commits or rolls back pending updates only when the transaction associated with that cursor ends.
- A transaction commit or abort has no effect on SQL cursor statements that do not manipulate result rows, such as **declare cursor**, **open cursor**, **close cursor**, **set cursor rows**, and **deallocate cursor.** For example, if the client opens a cursor within a transaction and the

transaction aborts, the cursor remains open after the abort (unless the **close on endtran** is set or the chained mode is used).

Errors and Transaction Rollbacks

Transactions maintain the integrity of your data, and certain errors that would affect data integrity affect the state of implicit or explicit transactions:

- Errors with severity levels of 19 or greater

  Since these errors terminate the user connection to the server, any errors of level 19 or greater that occur while a user transaction is in progress abort the transaction and roll back all statements to the outermost **begin transaction**. SQL Server always rolls back any uncommitted transactions at the end of a session.

- Data modification commands that affect data integrity:
  - Arithmetic overflow / divide by zero errors (effects on transactions can be changed with the **set arithabort arith_overflow** command)
  - Permissions violations
  - Rules violations
  - Duplicate key violations

The following table summarizes how a rollback caused by a permission or arithmetic error affects SQL Server processing in several different contexts (such as within a transaction, stored procedure, or trigger):

Context	Effect of Permission or Arithmetic Overflow Error Rollback
Transaction only	All data modifications since the start of the transaction are rolled back. If a transaction spans multiple batches, rollback affects all those batches. Any remaining commands in the batch are not executed. Processing resumes at the next batch.
Stored procedure only	Current command is aborted. Any previous commands are not rolled back. With permission errors, any following commands are executed. With **arithabort arith_overflow on**, any remaining commands in the batch are not executed; processing resumes at the next batch.

Table 3-33: How rollbacks from errors affect processing

Context	Effect of Permission or Arithmetic Overflow Error Rollback
Stored procedure in a transaction	Stored procedure is aborted. All data modifications since the start of the transaction are rolled back. If a transaction spans multiple batches, rollback affects all those batches. Any remaining commands in the batch are not executed. Processing resumes at the next batch.
Trigger only	Trigger is aborted, and trigger effects are rolled back. Any remaining commands in the batch are not executed. Processing resumes at the next batch.
Trigger in a transaction	Trigger is aborted, and trigger effects are rolled back. All data modifications since the start of the transaction are rolled back. If a transaction spans multiple batches, rollback affects all those batches. Any remaining commands in the batch are not executed. Processing resumes at the next batch.
Nested trigger	Inner trigger is aborted, and all trigger effects are rolled back. Any remaining commands in the batch are not executed. Processing resumes at the next batch.
Nested trigger in a transaction	Inner trigger is aborted, and all trigger effects are rolled back. All data modifications since the start of the transaction are rolled back. If a transaction spans multiple batches, rollback affects all those batches. Any remaining commands in the batch are not executed. Processing resumes at the next batch.
Trigger with rollback followed by an error in the transaction	Trigger effects are rolled back. All data modifications since the start of the transaction are rolled back. If a transaction spans multiple batches, rollback affects all those batches. Trigger continues and gets permission or arithabort error. Normally, that error halts trigger execution, but trigger continues in this case. After trigger completes, any remaining commands in the batch are not executed. Processing resumes at the next batch.

Table 3-33: How rollbacks from errors affect processing (continued)

Duplicate key errors and rules violations are special cases. The trigger completes (unless there is also a **return** statement) and statements such as **print**, **raiserror**, or remote procedure calls are performed. Then, the trigger and the rest of the transaction are rolled back and the rest of the batch is aborted. Note that remote procedure calls executed from inside a normal SQL transaction (not using DB-Library two-phase commit) are not rolled back by a **rollback** statement.

The following table summarizes how a rollback caused by a duplicate key or rules error affects SQL Server processing in several different contexts:

Context	Effect of Duplicate Key or Rules Error Rollback
Transaction only	Current command is aborted. Any previous commands are not rolled back, and any following commands are executed.
Stored procedure only	Same as above.
Stored procedure in a transaction	Same as above.
Trigger only	Trigger completes, but trigger effects are rolled back. Any remaining commands in the batch are not executed. Processing resumes at the next batch.
Trigger in a transaction	Trigger completes, but trigger effects are rolled back. All data modifications since the start of the transaction are rolled back. If a transaction spans multiple batches, rollback affects all those batches. Any remaining commands in the batch are not executed. Processing resumes at the next batch.
Nested trigger	Inner trigger completes, but all trigger effects are rolled back. Any remaining commands in the batch are not executed. Processing resumes at the next batch.

Table 3-34: How rollbacks from duplicate key or rules errors affect processing

Context	Effect of Duplicate Key or Rules Error Rollback
Nested trigger in a transaction	Inner trigger completes, but all trigger effects are rolled back. All data modifications since the start of the transaction are rolled back. If a transaction spans multiple batches, rollback affects all those batches. Any remaining commands in the batch are not executed. Processing resumes at the next batch.
Trigger with rollback followed by an error in the transaction	Trigger effects are rolled back. All data modifications since the start of the transaction are rolled back. If a transaction spans multiple batches, rollback affects all those batches. Trigger continues and gets duplicate key or rules error. Normally, the trigger rolls back effects and continues, but trigger effects are not rolled back in this case. After trigger completes, any remaining commands in the batch are not executed. Processing resumes at the next batch.

Table 3-34: How rollbacks from duplicate key or rules errors affect processing (continued)

See Also

Commands	**begin transaction, commit, rollback, set**

Variables (Local and Global)

The Two Kinds of Variables

Variables are defined entities that are assigned values. SQL Server has two kinds of variables:

- A **local variable** is defined with a **declare** statement and assigned an initial value within the statement batch where it is declared with a **select** statement.
- **Global variables** are SQL Server-supplied variables that have system-supplied values.

Where to Use Local Variables

Variables can be used nearly any place where the SQL syntax indicates that an expression can be used, such as *char_expr*, *integer_expression*, *numeric_expr* or *float_expr*. They can be passed as parameters to system procedures. Variables are often used in a batch or procedure as counters for **while** loops or **if-else** blocks.

Declaring a Local Variable

Use the following syntax to declare a local variable's name and datatype:

```
declare @variable_name datatype
   [, @variable_name datatype]...
```

The variable name must be preceded by the "@" sign and conform to the rules for identifiers. The datatype can be any datatype except *text*, *image*, or *sysname*.

Assigning a Value to a Local Variable

When a variable is declared, it has NULL value. Use a **select** statement to assign a specific value to the variable:

```
select @variable_name = expression
       [ , @variable_name = expression ]...
   [from clause] [where clause] [group by clause]
   [having clause] [order by clause] [compute clause]
```

The **select** statement that assigns values to local variables should return a single value. If it returns more than one value, the variable is assigned the last value returned. If it does not return a value (for example, if no rows are matched in a **where** clause) the value of the variable is left unchanged.

You cannot use the same **select** statement to assign a value to a variable and to retrieve data. For example, the following is not legal:

```
/* ILLEGAL STATEMENT */
declare @veryhigh money
select @veryhigh = max(price), title_id from titles
```

Do not use a single **select** statement to assign a value to one variable and then to another whose value is based on the first. Doing so can yield unpredictable results. For example, the following queries both try to find the value of *@c2*. The first query yields NULL, while the second query yields the correct answer, 0.033333:

```
/* this is wrong*/
declare @c1 float, @c2 float
select @c1 = 1000/1000, @c2 = @c1/30
select @c1, @c2

/* do it this way */
declare @c1 float, @c2 float
select @c1 = 1000/1000
select @c2 = @c1/30
select @c1 , @c2
```

Examples of Local Variable Usage

1.
```
declare @veryhigh money
select @veryhigh = max(price) from titles
if @veryhigh > $20
    print "Ouch!"
```

2.
```
declare @one varchar(18), @two varchar(18)
select @one = "this is one", @two = "this is two"
if @one = "this is one"
    print "you got one"
if @two = "this is two"
    print "you got two"
else print "nope"
```

3.
```
 /* Determine if a given au_id has a row in au_pix*/
/* Turn off result counting */
set nocount on
/* declare the variables */
declare @c int,
        @min_id varchar(30)
/*First, count the rows*/
select @c = count(*) from authors
/* Initialize @min_id to “” */
select @min_id = “”
/* while loop executes once for each authors row */
```

```
while @c > 0
begin
    /*Find the smallest au_id*/
    select @min_id = min(au_id)
        from authors
        where au_id > @min_id
    /*Is there a match in au_pix?*/
    maxreadif exists (select au_id
        from au_pix
        where au_id = @min_id)
    begin
        print "A Match! %1!", @min_id
    end
    select @c = @c -1  /*decrement the counter */
end
```

This example uses local variables in a counter in a **while** loop, for doing matching in a **where** clause, in an **if** statement, and also sets or resets values in **select** statements.

Using Global Variables

Predefined global variables are distinguished from local variables by having two @ signs preceding their names, for example, *@@error*, *@@rowcount*. Users cannot create global variables, and cannot update the value of global variables directly in a **select** statement.

Many global variables report on system activity occurring from the last time SQL Server was started. The system procedure **sp_monitor** displays the current values of some of the global variables.

List of Global Variables

The global variables are:

@@char_convert - contains 0 if character set conversion is not in effect. Contains 1 if character set conversion is in effect.

@@client_csname - client's character set name. Set to NULL if client character set has never been initialized; otherwise, it contains the name of the most recently used character set.

@@client_csid - client's character set ID. Set to -1 if client character set has never been initialized; otherwise, it contains the most recently used client character set ID from *syscharsets*.

@@connections - number of logins or attempted logins since SQL Server was last started.

@@cpu_busy - amount of time, in ticks, that the CPU has spent doing SQL Server work since the last time SQL Server was started.

@@error - commonly used to check the error status (succeeded or failed) of the most recently executed statement. It contains 0 if the previous transaction succeeded; otherwise, it contains the last error number generated by the system. A statement such as:

```
if @@error != 0 return
```

causes an exit if an error occurs.

Every Transact-SQL statement resets *@@error*, including **print** statements or **if** tests, so the status check must immediately follow the statement whose success is in question.

@@identity - last value inserted into an IDENTITY column by an **insert** or **select into** statement. *@@identity* is reset each time a row is inserted into a table. If a statement inserts multiple rows, @@identity reflects the IDENTITY value for the last row inserted. If the affected table does not contain an IDENTITY column, *@@identity* is set to 0.

The value of *@@identity* is not affected by the failure of an **insert** or **select into** statement, or the rollback of the transaction that contained it. *@@identity* retains the last value inserted into an IDENTITY column, even if the statement that inserted it fails to commit.

@@idle - amount of time, in ticks, that SQL Server has been idle since it was last started.

@@io_busy - amount of time, in ticks, that SQL Server has spent doing input and output operations since it was last started.

@@isolation - current isolation level of the Transact-SQL program. *@@isolation* takes the value of the active level (1 or 3).

@@langid - defines the local language id of the language currently in use (specified in *syslanguages.langid*).

@@language - defines the name of the language currently in use (specified in *syslanguages.name*).

@@maxcharlen - maximum length, in bytes, of a character in SQL Server's default character set.

@@max_connections - maximum number of simultaneous connections that can be made with SQL Server in this computer environment. The user can configure SQL Server for any number

of connections less than or equal to the value of *@@max_connections* with **sp_configure "user connections"**.

@@ncharsize - average length, in bytes, of a national character.

@@nestlevel - nesting level of current execution (initially 0). Each time a stored procedure or trigger calls another stored procedure or trigger, the nesting level is incremented. If the maximum of 16 is exceeded, the transaction aborts.

@@pack_received - number of input packets read by SQL Server since it was last started.

@@pack_sent - number of output packets written by SQL Server since it was last started.

@@packet_errors - number of errors that have occurred while SQL Server was sending and receiving packets.

@@procid - stored procedure ID of the currently executing procedure.

@@rowcount - number of rows affected by the last command. *@@rowcount* is set to zero by any command which does not return rows, such as an **if** statement. With cursors, *@@rowcount* represents the cumulative number of rows returned from the cursor result set to the client, up to the last **fetch** request.

@@servername - name of the local SQL Server. You must define a server name with **sp_addserver**, and then restart SQL Server.

@@spid - server process ID number of the current process.

@@sqlstatus - contains status information resulting from the last **fetch** statement. *@@sqlstatus* may contain the following values:

Value	Meaning
0	The **fetch** statement completed successfully.
1	The **fetch** statement resulted in an error.
2	There is no more data in the result set. This warning occurs if the current cursor position is on the last row in the result set and the client submits a **fetch** command for that cursor.

Table 3-35: @@sqlstatus Values

@@textsize - current value of the **set textsize** option, which specifies the maximum length, in bytes, of text or image data to be returned with a **select** statement. Defaults to 32K.

@@thresh_hysteresis - change in free space required to activate a threshold. This amount, also known as the hysteresis value, is measured in 2K database pages. It determines how closely thresholds can be placed on a database segment.

@@timeticks - number of microseconds per tick. The amount of time per tick is machine dependent.

@@total_errors - number of errors that have occurred while SQL Server was reading or writing.

@@total_read - number of disk reads by SQL Server since it was last started.

@@total_write - number of disk writes by SQL Server since it was last started.

@@tranchained - current transaction mode of the Transact-SQL program. *@@tranchained* returns a 0 for unchained or a 1 for chained.

@@trancount - nesting level of transactions. Each **begin transaction** in a batch increments the transaction count. When you query *@@trancount* in chained transaction mode, its value is never zero since the query automatically initiates a transaction.

@@transtate - current state of a transaction after a statement executes. However, *@@transtate* does not get cleared for each statement, unlike *@@error*. *@@transtate* may contain the following values:

Value	Meaning
0	Transaction in progress: an explicit or implicit transaction is in effect; the previous statement executed successfully.
1	Transaction succeeded: the transaction completed and committed its changes.
2	Statement aborted: the previous statement was aborted; no effect on the transaction.
3	Transaction aborted: the transaction aborted and rolled back any changes.

Table 3-36: @@transtate Values

@@version - date of the current version of SQL Server.

See Also

Commands	**declare**, **print**, **raiserror**
Topics	Datatypes
System procedures	**sp_monitor**

Wildcard Characters

Function

Represent one or more characters, or a range of characters, in a match string.

Using Wildcards

Use wildcards in **where** and **having** clauses to find character or date/time information that is **like**—or **not like**—the match string:

```
{where | having} [not]
   expression [not] like match_string
       [escape "escape_character"]
```

expression can be any combination of column names, constants, or functions with a character value.

match_string is the pattern to be found in the expression. It can be a any combination of constants, variables, and column names or a concatenated expression, such as:

```
like @variable + "%".
```

If the match string is a constant, it must always be enclosed in single or double quotes.

The Percent (%) Wildcard

Use the % wildcard to represent any string of zero or more characters. For example, to find all the phone numbers in the *authors* table that begin with the 415 area code:

```
select phone
from authors
where phone like "415%"
```

To find names that have the characters “en” in them (Bennet, Green, McBadden):

```
select au_lname
from authors
where au_lname like "%en%"
```

Trailing blanks following “%” in a like clause are truncated to a single trailing blank. For example, “%” followed by 2 spaces matches “X ”(one space); “X ” (two spaces); “X   ” (three spaces) or any number of trailing spaces.

The Underscore (_) Wildcard

Use the _ wildcard to represent any single character. For example, to find all six-letter names that end with “heryl” (Cheryl):

```
select au_fname
from authors
where au_fname like "_heryl"
```

Bracketed ([ ]) Characters

Use brackets to enclose a range of characters, such as [a-f], or a set of characters such as [a2Br]. When ranges are used, all values in the sort order between (and including) *rangespec1* and *rangespec2* are returned. For example, [0-z] matches 0-9, A-Z and a-z (and several punctuation characters) in 7-bit ASCII.

To find names ending with “inger” and beginning with any single character between M and Z:

```
select au_lname
from authors
where au_lname like "[M-Z]inger"
```

To find both DeFrance and deFrance:

```
select au_lname
from authors
where au_lname like "[dD]eFrance"
```

The Caret (^) Wildcard

The caret is the negative wildcard. Use it to find strings that do not match a particular pattern. For example, [^a-f] finds strings that are not in the range a-f and [^a2bR] finds strings that are not a, 2, b, or R.

To find names beginning with M where the second letter is not c:

```
select au_lname
from authors
where au_lname like "M[^c]%"
```

When ranges are used, all values in the sort order between (and including) *rangespec1* and *rangespec2* are returned. For example, [0-z] matches 0-9, A-Z , a-z, and several punctuation characters in 7-bit ASCII.

Using not like

Use **not like** to find strings that do not match a particular pattern. These two queries are equivalent: they find all the phone numbers in the *authors* table that do not begin with the 415 area code.

```
select phone
from authors
where phone not like "415%"

select phone
from authors
where not phone like "415%"
```

Different Results from not like and ^

You cannot always duplicate **not like** patterns with **like** and the negative wildcard [^]. This is because **not like** finds the items that do not match the entire **like** pattern, while **like** with negative wildcards is evaluated one character at a time.

For example, this query finds the system tables in a database whose names begin with "sys":

```
select name
from sysobjects
where name like "sys%"
```

To see all the objects that are **not** system tables, use

```
not like "sys%"
```

If you have a total of 32 objects and the **like** finds 13 names that match the pattern, the **not like** will find exactly the 19 objects that do not match the **like** pattern.

You won't always find the same names with a pattern such as **like** "[^s][^y][^s]%"—instead of 19, you might get only 14, with all the names that begin with "s" **or** have "y" as the second letter **or** have "s" as the third letter eliminated from the results, as well as the system table names. This is because match strings with negative wildcards are evaluated in steps, one wildcard at a time. If the match fails at any point in the evaluation, it is eliminated.

Case and Accent Insensitivity

If your SQL Server uses a case-insensitive sort order, case is ignored when comparing *expression* and *match_string*. For example, this clause:

```
where col_name like "Sm%"
```

would return "Smith", "smith" and "SMITH" on a case-insensitive SQL Server.

If your SQL Server is also accent-insensitive, it will treat all accented characters as equal to each other and to their unaccented counterparts, both uppercase and lowercase. The **sp_helpsort** system

procedure displays the characters that are treated as equivalent, displaying an "=" between them.

Using Multibyte Wildcard Characters

If the multibyte character set configured on your SQL Server defines equivalent double-byte characters for the wildcards _, %, - [,], and ^, you can substitute the equivalent character in the match string. The underscore equivalent represents either a single- or double-byte character in the match string.

Using Wildcards as Literal Characters

There are two ways to use %, _, [ ], or [^] as literal characters in a **like** match string rather than as wildcards:

- Square brackets (Transact-SQL extension)
- The **escape** clause (ANSI compliant)

Using Square Brackets as Escape Characters

Use square brackets as escape characters for the percent sign, the underscore, and the open bracket. The close bracket does not need an escape character; use it by itself. To use the dash as a literal character, it must be the first character inside a set of square brackets.

***like* Predicate**	**Meaning**
`like "5%"`	5 followed by any string of 0 or more characters
`like "5[%]"`	5%
`like "_n"`	an, in, on, etc.
`like "[_]n"`	_n
`like "[a-cdf]"`	a, b, c, d, or f
`like "[-acdf]"`	-, a, c, d, or f
`like "[[]"`	[
`like "]"`	]
like "[[]ab]"	[]ab

Table 3-37: Using square brackets to search for wildcards

Using the escape Clause

Use the **escape** clause to specify an escape character. Any single character in the server's default character set can be used as an escape character. If you try to use more than one character as an escape character, SQL Server generates an exception.

Do not use existing wildcard characters as escape characters because:

- If you specify the underscore (_) or percent sign (%) as escape characters, they lose their special meaning within that **like** predicate, and act only as escape characters.
- If you specify the open or close bracket ([or]) as an escape character, the Transact-SQL meaning of the bracket is disabled within that **like** predicate.
- If you specify the dash or caret (- or ^) as escape characters, they lose the special meaning that they normally have within square brackets, and act only as escape characters.

An escape character retains its special meaning within square brackets, unlike the wildcard characters such as the underscore, the percent sign, and the open bracket.

The escape character is valid only within its **like** predicate and has no effect on other **like** predicates contained in the same statement. The only characters that are valid following an escape character are the wildcard characters (_, %, [,], or [^]), and the escape character itself. The escape character affects only the character following it, and subsequent characters are not affected by it.

If the pattern contains two literal occurrences of the character that happens to be the escape character, the string must contain four consecutive escape characters. If the escape character does not divide the pattern into pieces of one or two characters, SQL Server returns an error message.

Following are examples of **like** predicates with **escape** clauses:.

***like* Predicate**	**Meaning**
`like "5@%" escape "@"`	5%
`like "*_n" escape "*"`	_n
`like "%80@%%" escape "@"`	string containing 80%
`like "*_sql**%" escape "*"`	string containing _sql*
`like "%#####_#%%" escape "#"`	string containing ##_%

Table 3-38: Using the escape clause

To enforce ANSI behavior and disable the special meaning of the square brackets, use **set fipsflagger on**.

Wildcards Have No Special Meaning Without like

Wildcards used without **like** have no special meaning. For example, this query finds any phone numbers that start with the four characters 415%:

```
select phone
from authors
where phone = "415%"
```

Using Wildcards with *datetime* Data

When you use **like** with *datetime* values, SQL Server converts the dates to the standard *datetime* format, and then to *varchar*. Since the standard storage format doesn't include seconds or milliseconds, you cannot search for seconds or milliseconds with **like** and a pattern.

It is a good idea to use **like** when you search for *datetime* values, since *datetime* entries may contain a variety of date parts. For example, if you insert the value "9:20" and the current date into a column named *arrival_time*, the clause:

```
where arrival_time = '9:20'
```

would not find it because SQL Server converts the entry into "Jan 1 1900 9:20AM." However, the clause below would find it:

```
where arrival_time like '%9:20%'
```

See Also

Commands	**where** Clause

Appendixes

A Reserved Words

Keywords are words that have a special meaning. This appendix lists Transact-SQL, APT-SQL, and SQL92 keywords.

Transact-SQL Reserved Words

The following words are reserved by SQL Server as keywords (command verbs) and cannot be used for the names of database objects such as databases, tables, rules, and defaults. Reserved words can be used for the names of local variables and for stored procedure parameter names.

add	close	disk	from
all	clustered	distinct	goto
alter	commit	double	grant
and	compute	drop	group
any	confirm	dummy	having
arith_overflow	constraint	dump	holdlock
as	continue	else	identity
asc	controlrow	end	identity_insert
at	convert	endtran	if
authorization	count	errlvl	in
avg	create	errorexit	index
begin	current	escape	insert
between	cursor	except	intersect
break	data_pgs	exec	into
browse	database	execute	is
bulk	dbcc	exists	isolation
by	deallocate	exit	key
cascade	declare	fetch	kill
char_convert	default	fillfactor	level
check	delete	for	like

Table A-1: Transact-SQL reserved words

checkpoint	desc	foreign	lineno
load	perm	rollback	to
max	permanent	rowcnt	tran
min	plan	rowcount	transaction
mirror	precision	rows	trigger
mirrorexit	prepare	rule	truncate
national	primary	save	tsequal
noholdlock	print	schema	union
nonclustered	privileges	select	unique
not	proc	set	update
null	procedure	setuser	used_pgs
numeric_truncation	processexit	shared	user
of	public	shutdown	user_option
off	raiserror	some	using
offsets	read	statistics	values
on	readtext	stripe	varying
once	reconfigure	sum	view
only	references	syb_identity	waitfor
open	replace	syb_restree	where
option	reserved_pgs	table	while
or	return	temp	with
order	revoke	temporary	work
over	role	textsize	writetext

Table A-1: Transact-SQL reserved words (continued)

APT-SQL Keywords

Table A-2: APT-SQL keywords lists the APT-SQL keywords which are not reserved words in Transact-SQL. If you are planning to use APT-SQL, avoid using these words as identifiers.

$channel	charindex	int	smallint
$curfield	closesql	interruptsql	sqlbegin
$curform	connect	list	sqlend
$curgroup	curindex	local	sqlexpr
$curpick	cursqlindex	log	sqlrow
$date	datalength	lower	submit
$index	datename	mchoice	substring
$status	datepart	menu	switch
abort	datetime	menubar	switchend
and	define	money	system
append	disconnect	nextquery	tab
apt	enter	nomsg	text
backtab	entry	opensql	textport
bell	exitform	parentname	tinyint
binary	exp	perform	trace
bit	false	positionform	transfer
call	fetchsql	post	trim
callextern	field	printform	true
callform	float	rchoice	upper
callreport	foreach	remote	useform
cancelform	form	reset	variable
case	global	schoice	
channel	hidden	scroll	
char	image	shared	

Table A-2: APT-SQL keywords

SQL92 Keywords

SQL Server 10.0 includes entry-level SQL92 features. Full SQL92 implementation includes the words listed in the following tables as command syntax. Since upgrading identifiers can be a complex process, we are providing this list for your convenience. The publication of this information does not commit Sybase to providing all of these SQL92 features in subsequent releases, and in addition subsequent releases may include keywords not included in this list.

Table A-3 lists the SQL92 keywords which are not reserved words in Transact-SQL.

absolute	constraints	false
action	corresponding	first
allocate	cross	float
are	current_date	found
assertion	current_time	full
bit	current_timestamp	get
bit_length	current_user	global
both	date	go
cascaded	day	hour
case	dec	immediate
cast	decimal	indicator
catalog	deferrable	initially
char	deferred	inner
character	describe	input
char_length	descriptor	insensitive
character_length	diagnostics	int
coalesce	disconnect	integer
collate	domain	interval
collation	end-exec	join
column	exception	language
connect	external	last
connection	extract	leading

Table A-3: SQL92 keywords that are not Transact-SQL reserved words

left	preserve	time
local	prior	timestamp
lower	real	timezone_hour
match	relative	timezone_minute
minute	restrict	trailing
module	right	translate
month	scroll	translation
names	second	trim
natural	section	true
nchar	session	unknown
next	session_user	upper
no	size	usage
nullif	smallint	value
numeric	space	varchar
octet_length	sql	when
outer	sqlcode	whenever
output	sqlerror	write
overlaps	sqlstate	year
pad	substring	zone
partial	system_user	
position	then	

Table A-3: SQL92 keywords that are not Transact-SQL reserved words (continued)

Potential SQL92 Reserved Words

If you are using the ISO / IEC 9075:1989 standard, also avoid using the words in *Table A-4: Potential reserved words*, as these words may become SQL92 reserved words in the future.

after	modify	routine
alias	new	row
async	none	savepoint
before	object	search
boolean	oid	sensitive
breadth	old	sequence
completion	operation	signal
call	operators	similar
cycle	others	sqlexception
data	parameters	structure
depth	pendant	test
dictionary	preorder	there
each	private	type
elseif	protected	under
equals	recursive	variable
general	ref	virtual
ignore	referencing	visible
leave	resignal	wait
less	return	without
limit	returns	
loop	role	

Table A-4: Potential reserved words

B The System Tables

Introduction

All of the tables in the *master* database are system tables. Some of these tables also occur in user databases—they are automatically created when the **create database** command is issued.

These system tables occur in all databases:

System Table	Contents
sysalternates	One row for each SQL Server user mapped to a database user
syscolumns	One row for each column in a table or view, and for each parameter in a procedure
syscomments	One or more rows for each view, rule, default, trigger, and procedure, giving SQL definition statement
sysconstraints	One row for each referential and check constraint associated with a table or column
sysdepends	One row for each procedure, view, or table that is referenced by a procedure, view, or trigger
sysindexes	One row for each clustered or nonclustered index, and one row for each table with no indexes, and an additional row for each table containing *text* or *image* data.
syskeys	One row for each primary, foreign, or common key; set by user (not maintained by SQL Server)
syslogs	Transaction log
sysobjects	One row for each table, view, procedure, rule, trigger default, log, and (in *tempdb* only) temporary object
sysprocedures	One row for each view, rule, default, trigger, and procedure, giving internal definition
sysprotects	User permissions information
sysreferences	One row for each referential integrity constraint declared on a table or column
sysroles	Maps server-wide roles to local database groups
syssegments	One row for each segment (named collection of disk pieces)

Table B-1: System tables that occur in all databases

System Table	Contents
systhresholds	One row for each threshold defined for the database
systypes	One row for each system-supplied and user-defined datatype
sysusermessages	One row for each user-defined message
sysusers	One row for each user allowed in the database

Table B-1: System tables that occur in all databases (continued)

These system tables occur in the *master* database only:

System Table	Contents
syscharsets	One row for each character set or sort order
sysconfigures	One row for each user-settable configuration parameter
syscurconfigs	Information about configuration parameters currently being used by SQL Server
sysdatabases	One row for each database on SQL Server
sysdevices	One row for each tape dump device, disk dump device, disk for databases, and disk partition for databases
sysengines	One row for each SQL Server engine currently on line
syslanguages	One row for each language (except U.S. English) known to the server
syslocks	Information about active locks
sysloginroles	One row for each server login that possesses a system-defined role
syslogins	One row for each valid SQL Server user account
sysmessages	One row for each system error or warning
sysprocesses	Information about server processes
sysremotelogins	One row for each remote user
syssrvroles	One row for each server-wide role
sysservers	One row for each remote SQL Server
sysusages	One row for each disk piece allocated to a database

Table B-2: System tables that occur in the master database only

These system tables occur in the *sybsecurity* database only:

System Table	Contents
sysaudits	One row for each audit record
sysauditoptions	One row for each global audit option

Table B-3: System tables that occur in the sybsecurity *database only*

In the pages that follow, each system table is described in more detail, including a list of their columns and datatypes. In addition, the indexes and the system procedures that reference a particular table are listed.

The word "reserved" in the column description means that the column is currently not being used by SQL Server.

Permissions for use of the system tables can be controlled by the database owner, just like permissions on any other tables.

The SYBASE installation program sets up permissions so that all users can read the system tables, with the exception of a few fields. (See the *SQL Server Installation Guide* for details.)

All direct updates on system tables are by default not allowed even for the database owner. Instead, SQL Server supplies system procedures to make any normally needed updates and additions to system tables.

You can allow direct updates to the system tables if it becomes necessary to modify them in a way that cannot be accomplished with a system procedure. To accomplish this, a System Security Officer must reset the configuration variable called **allow updates** with the system procedure **sp_configure**, and a System Administrator must then execute the **reconfigure** command. For information, see the *System Administration Guide*.

There are entries in some of the *master* database tables that should not be altered by any user under any circumstances. For example, do not attempt to modify *syslogs* with a **delete**, **update**, or **insert** command. In addition, an attempt to **delete** all rows from *syslogs* will put SQL Server into an infinite loop that eventually fills up the entire database.

Note that aggregate functions cannot be used on virtual tables such as *syslocks* and *sysprocesses*.

A diagram of the system tables and their relationships is included at the back of the bound *System Administration Guide* and the *SQL ServerReference Manuals* Volumes 1 and 2. The diagram is not included with camera-ready copy or CD-ROM versions of the manuals.

sysalternates

(all databases)

Description

sysalternates contains one row for each SQL Server user mapped (or aliased) to a user of the current database. When a user tries to access a database, SQL Server looks for a valid *uid* entry in *sysusers*. If none is found, it looks in *sysalternates.suid*. If the user's *suid* is found there, he or she is treated as the database user whose *suid* is listed in *sysalternates.altsuid*.

On the SQL Server distribution tape, there are no entries in *sysalternates*.

Column	Datatype	Description
suid	*smallint*	Server user ID of user being mapped
altsuid	*smallint*	Server user ID of user to whom another user is mapped

Table B-4: Columns in the sysalternates *table*

Indexes

Unique clustered index on *suid*

Referenced by System Procedures

sp_addalias, sp_adduser, sp_changedbowner, sp_dropalias, sp_dropuser, sp_helpuser

sysauditoptions

(*sybsecurity* database)

Description

sysauditoptions contains one row for each global audit option (options set via **sp_auditoption**).These are the system-wide options only, and do not include database, object, stored procedure, trigger, and user audit options. The default value for each option is 0 or "off." *sysauditoptions* can be accessed only by System Security Officers.

Column	Datatype	Description
optn	*smallint*	Option number. See Table B-6.
value	*smallint*	Current value; one of the following: **off** = 0 **ok** = 1 **fail** = 2 **both** = 3 (where applicable) For error auditing (*optn*=13), the values are: **off** = 0 **nonfatal** = 1 **fatal** = 2 **both** = 3
min	*smallint*	Minimum valid value for this option
max	*smallint*	Maximum valid value for this option
name	*varchar(30)*	Name of option
svalue	*varchar(30)*	String equivalent of the current value: for example, "on", "off", "nonfatal"
comment	*varchar(255)*	Description of option

Table B-5: Columns in the sysauditoptions *table*

Possible values for *optn* are:

Option Number	Description
1	Enable or disable auditing
2	Unused
3	Login auditing

Table B-6: Audit option values and descriptions

Option Number	Description
4	Logout auditing
5	Server boot auditing
6	RPC connection auditing
7	Auditing use of the **set** command to turn roles on and off
8	Auditing commands requiring **sa_role** role
9	Auditing commands requiring **sso_role** role
10	Auditing commands requiring **oper_role** role
12	Auditing commands requiring **navigator** role
13	Error auditing
14	Ad hoc auditing
15	Auditing commands requiring **replication** role

Table B-6: Audit option values and descriptions (continued)

Indexes

None

Referenced by System Procedures

sp_auditoption

sysaudits

(*sybsecurity* database)

Description

The *sysaudits* table contains one row for each audit record.

Column	Datatype	Description
event	*smallint*	Type of event being audited. See Table B-8.
eventmod	*smallint*	Further information about the event. Possible values are: 0 = no modifier for this event 1 = successful occurrence of this event; for error auditing (*event*=13), a nonfatal error 2 = failed occurrence of this event; for error auditing (*event*=13), a fatal error
spid	*smallint*	Server process ID of the process that caused the audit record to be written
eventtime	*datetime*	Date and time of the audited event
sequence	*smallint*	Sequence number of the record within a single event; some events require more than one audit record
suid	*smallint*	Server login ID of the user who performed the audited event
dbid	*int null*	Database ID in which the audited event occurred or the object / stored procedure / trigger resides, depending on the type of event
objid	*int null*	ID of the accessed object or stored procedure / trigger
xactid	*binary(6) null*	ID of the transaction containing the audited event. For a multi-database transaction, this is the transaction ID from the database where the transaction originated.
loginname	*varchar(30) null*	Login name corresponding to the *suid*
dbname	*varchar(30) null*	Database name corresponding to the *dbid*
objname	*varchar(30) null*	Object name corresponding to the *objid*

Table B-7: Columns in the sysaudits *table*

Column	Datatype	Description
objowner	*varchar(30) null*	Name of the owner of *objid*
extrainfo	*varchar(255) null*	Additional information about the audited event; contents vary with the type of event audited. (See Table B-8.)

Table B-7: Columns in the sysaudits *table*

Possible values for the *event* column are shown in Table B-8, along with the corresponding contents of the *extrainfo* column. Global audit events have a code of less than 100. All other event types are numbered starting at 100.

event No.	Description	Contents of *extrainfo* Column
1	Enable auditing	NULL
2	Disable auditing	NULL
3	Login	Host name
4	Logout	Host name
5	Server boot	Names of the server program, master device, interfaces file path, server, and error log file
6	RPC connection	Remote server name, host name
7	Use of **set** command to turn roles on and off	Role, new setting
8	Command requiring **sa_role** role	Command type
9	Command requiring **sso_role** role	Command type
10	Command requiring **oper_role** role	Command type
12	Command requiring **navigator** role	Command type
13	Error	Error number, severity, and state
14	Ad hoc audit record	User-supplied comment text
15	Command requiring **replication** role	Command type
100	Database reference	Command type
101	Table reference	Command type
102	View reference	Command type
103	Stored procedure execution	Parameter list
104	Trigger execution	NULL
105	User's attempts to access a table	Command type
106	User's attempt to access a view	Command type
107	User's command text auditing	Command batch text

Table B-8: Contents of event *and* extrainfo *columns of* sysaudits

Indexes

None

Referenced by System Procedures

None

syscharsets

(*master* database only)

Description

syscharsets contains one row for each character set and sort order defined for use by SQL Server. One of the sort orders is marked in *master..sysconfigures* as the default sort order, which is the only one actually in use.

Column	Datatype	Description
type	*smallint*	The type of entity this row represents. Numbers from 1001 to 1999 represent character sets. Numbers from 2000 to 2999 represent sort orders.
id	*tinyint*	The ID for a character set or sort order. A sort order is defined by the combination of the sort order ID and the character set ID (*csid*). The character set is defined by id, which must be unique. Sybase reserves ID numbers 0-200.
csid	*tinyint*	If the row represents a character set, this field is unused. If the row represents a sort order, this is the ID of the character set that sort order is built on. A character set row with this ID must exist in this table.
status	*smallint*	Internal system status information bits.
name	*varchar(30)*	A unique name for the character set or sort order. Must contain only the 7-bit ASCII letters A-Z or a-z, digits 0-9, and underscores (_), and begin with a letter.
description	*varchar(255)*	An optional description of the features of the character set or sort order.
definition	*image*	The internal definition of the character set or sort order. The structure of the data in this field depends on the *type*.

Table B-9: Columns in the syscharsets *table*

Indexes

unique clustered index on *id, csid, type*
unique nonclustered index on *name*

Referenced by System Procedures

sp_checkreswords, sp_helpsort, sp_serverinfo

syscolumns

(all databases)

Description

syscolumns contains one row for every column in every table and view, and a row for each parameter in a procedure.

Column	Datatype	Description
id	*int*	ID of table to which this column belongs or of procedure with which this parameter is associated
number	*smallint*	Sub-procedure number when the procedure is grouped (0 for non-procedure entries)
colid	*tinyint*	Column ID
status	*tinyint*	Indicates unique position for *bit* columns, whether NULL values are legal in this column, and if a check constraint exists for the column
type	*tinyint*	Physical storage type; copied from *systypes*
length	*tinyint*	Physical length of data; copied from *systypes* or supplied by user
offset	*smallint*	Offset into the row where this column appears, if negative, this is a variable-length column
usertype	*smallint*	User type ID; copied from *systypes*
cdefault	*int*	ID of the procedure that generates default value for this column
domain	*int*	Constraint ID of the first rule or check constraint for this column
name	*sysname*	Column name
printfmt	*varchar(255)*	Reserved
prec	*tinyint*	Number of significant digits
scale	*tinyint*	Number of digits to the right of the decimal point

Table B-10: Columns in the syscolumns *table*

Indexes

unique clustered index on *id, number, colid*

Referenced by System Procedures

sp_bindefault, sp_bindrule, sp_changegroup, sp_checknames, sp_checkreswords, sp_column_privileges, sp_columns, sp_commonkey, sp_droptype, sp_dropuser, sp_estspace, sp_foreignkey, sp_help, sp_helpconstraint, sp_helpjoins, sp_helprotect, sp_primarykey, sp_rename, sp_special_columns, sp_sproc_columns, sp_statistics, sp_unbindefault, sp_unbindrule

syscomments

(all databases)

Description

syscomments contains entries for each view, rule, default, trigger, table constraint, and procedure. The *text* field contains the original definition statements. If the *text* is longer than 255 bytes, the entries will span rows. Each object can occupy up to 65025 rows.

Column	Datatype	Description
id	*int*	Object ID to which this text applies
number	*smallint*	Sub-procedure number when the procedure is grouped (0 for non-procedure entries)
colid	*tinyint*	Sequence of 255 rows for the object
texttype	*smallint*	0 for system-supplied comment (for views, rules, defaults, triggers, and procedures). 1 for user-supplied comment (users can add entries that describe an object or column)
language	*smallint*	Reserved
text	*varchar(255)*	Actual text of SQL definition statement
colid2	*tinyint*	Indicates next sequence of rows for the object (see *colid* above); object can have up to 255 sequences of 255 rows each.

Table B-11: Columns in the syscomments *table*

Indexes

unique clustered index on *id, number, colid, texttype*

Referenced by System Procedures

sp_helpconstraint, sp_helptext

sysconfigures

(*master* database only)

Description

sysconfigures and *syscurconfigs* contain one row for each user-settable configuration variable.

Column	Datatype	Description
config	*smallint*	Configuration variable number
value	*int*	The user-modifiable value for the variable (being used by SQL Server only if **reconfigure** has been run)
comment	*varchar(255)*	Explanation of the configuration variable
status	*smallint*	Either 1 (dynamic, meaning variable takes effect when **reconfigure** is issued) or 0 (variable takes effect when SQL Server is restarted

Table B-12: Columns in the sysconfigures *table*

Here are the contents of *sysconfigures:*

config	value	comment	status
101	0	Maximum recovery interval in minutes	1
102	1	Allow updates to system tables	1
103	0	Number of user connections allowed	0
104	0	Size of available physical memory in 2k pages	0
105	0	Number of open databases allowed among all users	0
106	0	Number of locks for all users	0
107	0	Number of open database objects	0
108	0	Percentage of remaining memory used for procedure cache	0
109	0	Default fill factor percentage	0
110	0	Average time slice per process in milliseconds	0
111	0	Default database size in megabytes	0
112	0	Tape retention period in days	0
113	0	Recovery flags	0
115	1	Allow triggers to be invoked within triggers	1
116	0	Number of devices	0

Table B-13: Contents of sysconfigures

config	value	comment	status
117	1	Allow remote access	1
118	0	Number of remote logins	0
119	0	Number of remote sites	0
120	0	Number of remote connections	0
121	0	Number of pre-read packets per remote connection	0
122	1001	Upgrade version	1
123	50	Default sortorder ID	0
124	0	Default language	1
125	3	Language cache	0
126	1	Maximum online engines	0
127	1	Minimum online engines	0
128	0	Engine adjust interval	0
129	200	CPU accounting flush interval	1
130	1000	I/O accounting flush interval	1
131	1	Default character set ID	0
134	0	Stack size	0
135	0	System-wide password expiration interval	1
136	100	Audit queue size	0
137	0	Additional netmem	0
138	0	Default network packet size	0
139	0	Maximum network packet size	0
140	0	Number of extent I/O buffers	0
141	5000	Identity burning set factor	0

Table B-13: Contents of sysconfigures *(continued)*

Indexes

unique clustered index on *config*

Referenced by System Procedures

sp_configure

sysconstraints

(all databases)

Description

The *sysconstraints* table has one row for each referential and check constraint associated with a table or column.

Whenever a user declares a new check constraint or referential constraint using **create table** or **alter table**, SQL Server inserts a row into the *sysconstraints* table. The row remains until a user executes **alter table** to drop the constraint. Dropping a table by executing **drop table** removes all rows associated with that table from the *sysconstraints* table.

Column	Datatype	Description
colid	*tinyint*	Column number in the table
spare1	*tinyint*	Unused
constrid	*int*	Object ID of the constraint
tableid	*int*	ID of the table on which the constraint is declared
error	*int*	Constraint specific error message
status	*int*	The type of constraint: 0x0040 = a referential constraint 0x0080 = a check constraint
spare2	*int*	Unused

Table B-14: Columns in the sysconstraints *table*

Indexes

clustered index on *tableid, colid*
unique nonclustered index on *constrid*

Referenced by System Procedures

sp_bindmsg, sp_bindrule, sp_helpconstraint, sp_unbindmsg, sp_unbindrule

syscurconfigs

(*master* database only)

Description

syscurconfigs is built dynamically when queried. Its structure is identical to that of *sysconfigures*. It contains an entry for each of the configuration variables, as does *sysconfigures*, but with the current values rather than the default values. In addition, it contains four rows that describe the configuration structure.

Column	Datatype	Description
config	*smallint*	Configuration variable number
value	*int*	The user-modifiable value for the variable (being used by SQL Server only if **reconfigure** has been run)
comment	*varchar(255)*	Explanation of the configuration variable
status	*smallint*	Either 1 (dynamic, meaning variable takes effect when **reconfigure** is issued) or 0 (variable takes effect when SQL Server is restarted

Table B-15: Columns in the syscurconfigs *table*

Referenced by System Procedures

sp_configure, sp_helpsort, sp_serverinfo

sysdatabases

(*master* database only)

Description

sysdatabases contains one row for each database on SQL Server. When SQL Server is installed, *sysdatabases* contains entries for the *master* database, the *model* database, the *sybsystemprocs* database and the *temporary* database. If you have installed auditing, it also contains an entry for the *sybsecurity* database

Column	Datatype	Description
name	*sysname*	Name of the database
dbid	*smallint*	Database ID
suid	*smallint*	Server user ID of database creator
status	*smallint*	Control bits; those which the user can set with **sp_dboption** are marked "settable." See Table B-17.
version	*smallint*	Version of SQL Server code under which database was created
logptr	*int*	Pointer to transaction log
crdate	*datetime*	Creation date
dumptrdate	*datetime*	Date of the last **dump transaction**
status2	*intn*	Additional control bits. See Table B-18.
audflags	*intn*	Audit settings for database
deftabaud	*intn*	Bit-mask that defines default audit settings for tables
defvwaud	*intn*	Bit-mask that defines default audit settings for views
defpraud	*intn*	Bit-mask that defines default audit settings for stored procedures

Table B-16: Columns in the sysdatabases *table*

The bit representations for the *status* column are:

Decimal	Hex	Status
4	0x04	**select into/bulkcopy**; settable
8	0x08	**trunc log on chkpt**; settable
16	0x10	**no chkpt on recovery**; settable
32	0x20	crashed while loading database, instructs recovery not to proceed
256	0x100	database suspect; not recovered; cannot be opened or used; can only be dropped with **dbcc dbrepair**
512	0x200	**ddl in tran**; settable
1024	0x400	**read only**; settable
2048	0x800	**dbo use only**; settable
4096	0x1000	**single user**; settable
8192	0x2000	**allow nulls by default**; settable
16384	0x4000	*dbname* has changed

Table B-17: status *control bits in the* sysdatabases *table*

The bit representations for the *status2* column are:

Decimal	Hex	Status
1	0x01	**abort tran on log full**; settable
2	0x02	**no free space acctg**; settable
4	0x04	**auto identity**; settable

Table B-18: status2 *control bits in the* sysdatabases *table*

Indexes

unique clustered index on *name*
unique nonclustered index on *dbid*

Referenced by System Procedures

sp_addlogin, sp_addsegment, sp_addtype, sp_changedbowner, sp_checknames, sp_checkreswords, sp_databases, sp_dboption, sp_dbremap, sp_dropdevice, sp_dropsegment, sp_extendsegment, sp_helpdb, sp_logdevice, sp_renamedb, sp_tables

sysdepends

(all databases)

Description

sysdepends contains one row for each procedure, view, or table that is referenced by a procedure, view, or trigger.

Column	Datatype	Description
id	*int*	Object ID
number	*smallint*	Procedure number
depid	*int*	Dependent object ID
depnumber	*smallint*	Dependent procedure number
status	*smallint*	Internal status information
selall	*bit*	On if object is used in **select** * statement
resultobj	*bit*	On if object is being updated
readobj	*bit*	On if object is being read

Table B-19: Columns in the sysdepends *table*

Indexes

unique clustered index on *id, number, depid, depnumber*

Referenced by System Procedures

sp_depends

sysdevices

(*master* database only)

Description

sysdevices contains one row for each tape dump device, disk dump device, disk for databases, and disk partition for databases. On the SQL Server distribution tape, there are four entries in *sysdevices*: one for the master device (for databases), one for a disk dump device, and two for tape dump devices.

Column	Datatype	Description
low	*int*	First virtual page number on database device (not used for dump devices)
high	*int*	Last virtual page number on database device or dump device
status	*smallint*	Bit map indicating type of device, default and mirror status. See Table B-21.
cntrltype	*smallint*	Controller type (0 if database device, 2 if disk dump device or streaming tape, 3–8 if tape dump device)
name	*sysname*	Logical name of dump device or of database device
phyname	*varchar(127)*	Name of physical device
mirrorname	*varchar(127)*	Name of mirror device

Table B-20: Columns in the sysdevices *table*

The bit representations for the *status* column are additive. For example, “3” indicates a physical disk that is also a default.

The status control bits are:

Decimal	Hex	Status
1	0x01	default disk
2	0x02	physical disk
4	0x04	logical disk
8	0x08	skip header

Table B-21: status *control bits in the* sysdevices *table*

Decimal	Hex	Status
16	0x10	dump device
32	0x20	serial writes
64	0x40	device mirrored
128	0x80	reads mirrored
256	0x100	secondary mirror side only
512	0x200	mirror enabled
2048	0x800	used internally

Table B-21: status *control bits in the* sysdevices *table*

Indexes

unique clustered index on *name*

Referenced by System Procedures

sp_addsegment, sp_addumpdevice, sp_checknames, sp_checkreswords, sp_configure, sp_diskdefault, sp_dropdevice, sp_dropsegment, sp_extendsegment, sp_helpdb, sp_helpdevice, sp_helplog, sp_helpsegment, sp_logdevice, sp_volchanged

sysengines

(*master* database only)

Description

sysengines contains one row for each SQL Server engine currently on line.

Column	Datatype	Description
engine	*smallint*	Engine number
osprocid	*int*	Operating system process ID (may be NULL)
osprocname	*char*	Operating system process name (may be NULL)
status	*char*	One of: online, off-line, in create, in destroy, debug
affinitied	*int*	Number of SQL Server processes with affinity to this engine
cur_kpid	*int*	Kernel process ID of process currently running on this engine, if any
last_kpid	*int*	Kernel process ID of process which previously ran on this engine
idle_1	*tinyint*	Reserved
idle_2	*tinyint*	Reserved
idle_3	*tinyint*	Reserved
idle_4	*tinyint*	Reserved
starttime	*datetime*	Date and time engine came on line

Table B-22: Columns in the sysengines *table*

Indexes

none

Referenced by System Procedures

sp_monitor

sysindexes

(all databases)

Description

sysindexes contains one row for each clustered index, one row for each nonclustered index, one row for each table that has no clustered index, and one row for each table that contains *text* or *image* columns.

The column *doampg* is used only if the row describes a table or clustered index.

Column	Datatype	Description
name	*sysname*	Index or table name
id	*int*	ID of table, or ID of table to which index belongs
indid	*smallint*	0 if table, 1 if clustered index, >1 if nonclustered, 255 if text chain
doampg	*int*	Page number for the object allocation map of a table or clustered index
ioampg	*int*	Page number for the allocation map of a nonclustered index
oampgtrips	*int*	Ratio of OAM page to data page residency in cache
status2	*int*	Internal system status information. See Table B-24.
ipgtrips	*int*	Ratio of index page to data page residency in cache
first	*int*	Pointer to first data or leaf page
root	*int*	Pointer to root page if entry is an index; pointer to last page if entry is a table or text chain
distribution	*int*	Pointer to distribution page (if entry is an index)
usagecnt	*smallint*	Reserved
segment	*smallint*	Number of segment in which this object resides

Table B-23: Columns in the sysindexes *table*

Column	Datatype	Description
status	*smallint*	Internal system status information (See Table B-25)
rowpage	*smallint*	Maximum number of rows per page
minlen	*smallint*	Minimum size of a row
maxlen	*smallint*	Maximum size of a row
maxirow	*smallint*	Maximum size of a non-leaf index row
keycnt	*smallint*	Number of keys for a clustered index; number of keys+1 for a nonclustered index
keys1	*varbinary(255)*	Description of key columns (if entry is an index)
keys2	*varbinary(255)*	Description of key columns (if entry is an index)
soid	*tinyint*	Sort order ID that the index was created with. '0' if there is no character data in the keys
csid	*tinyint*	Character set ID that the index was created with. '0' if there is no character data in the keys

Table B-23: Columns in the sysindexes *table*

The bit representations for the *status2* column are:

Decimal	Hex	Status
1	0x1	Index part of referential integrity constraint
2	0x2	Index part of primary key / unique constraint

Table B-24: status2 *control bits in the* sysindexes *table*

The bit representations for the *status* column are:

Decimal	Hex	Status
1	0x1	Abort command or trigger if attempt to insert duplicate key
2	0x2	Unique index
4	0x4	Abort command or trigger if attempt to insert duplicate row
16	0x10	Clustered index
64	0x40	Index allows duplicate rows
32768	0x8000	Suspect index

Table B-25: status *control bits in the* sysindexes *table*

Indexes

unique clustered index on *id, indid*

Referenced by System Procedures

sp_checknames, sp_checkreswords, sp_dropsegment, sp_estspace, sp_help, sp_helpconstraint, sp_helpindex, sp_helplog, sp_helpsegment, sp_indsuspect, sp_pkeys, sp_placeobject, sp_rename, sp_spaceused, sp_special_columns, sp_statistics

syskeys

(all databases)

Description

syskeys contains one row for each primary, foreign, or common key.

Column	Datatype	Description
id	*int*	Object ID
type	*smallint*	Record type
depid	*int null*	Dependent object ID
keycnt	*int null*	The number of non-null keys
size	*int null*	Reserved
key1	*int null*	Column ID
key2	*int null*	Column ID
key3	*int null*	Column ID
key4	*int null*	Column ID
key5	*int null*	Column ID
key6	*int null*	Column ID
key7	*int null*	Column ID
key8	*int null*	Column ID
depkey1	*int null*	Column ID
depkey2	*int null*	Column ID
depkey3	*int null*	Column ID
depkey4	*int null*	Column ID
depkey5	*int null*	Column ID
depkey6	*int null*	Column ID
depkey7	*int null*	Column ID
depkey8	*int null*	Column ID

Table B-26: Columns in the syskeys *table*

Indexes

clustered index on *id*

Referenced by System Procedures

sp_commonkey, sp_dropkey, sp_foreignkey, sp_helpjoins, sp_helpkey, sp_primarykey

syslanguages

(*master* database only)

Description

syslanguages contains one row for each language known to SQL Server. *us_english* is not in *syslanguages*, but is always available to SQL Server.

Column	Datatype	Description
langid	*smallint*	Unique language ID
dateformat	*char(3)*	Date order, for example, "dmy"
datefirst	*tinyint*	First day of the week—1 for Monday, 2 for Tuesday, and so on, up to 7 for Sunday.
upgrade	*int*	SQL Server version of last upgrade for this language
name	*varchar(30)*	Official language name, for example, "french"
alias	*varchar(30)*	Alternate language name, for example, "francais"
months	*varchar(251)*	Comma-separated list of full-length month names, in order from January to December—each name is at most 20 characters long
shortmonths	*varchar(119)*	Comma-separated list of shortened month names, in order from January to December—each name is at most 9 characters long
days	*varchar(216)*	Comma-separated list of day names, in order from Monday to Sunday—each name is at most 30 characters long.

Table B-27: Columns in the syslanguages *table*

Indexes

unique clustered index on *langid*
unique nonclustered index on *name*
unique nonclustered index on *alias*

Referenced by System Procedures

sp_addlanguage, sp_addmessage, sp_checkreswords, sp_configure, sp_droplanguage, sp_dropmessage, sp_getmessage, sp_helplanguage, sp_setlangalias

syslocks

(*master* database only)

Description

syslocks contains information about active locks, but it is not a normal table. Rather, it is built dynamically when queried by a user. No updates to *syslocks* are allowed.

Column	Datatype	Description
id	*int*	Table ID
dbid	*smallint*	Database ID
page	*int*	Page number
type	*smallint*	Type of lock *(*bit values for the *type* column are listed in Table B-29)
spid	*smallint*	ID of process that holds the lock
class	*char(30)*	Name of the cursor this lock is associated with, if any

Table B-28: Columns in the syslocks *table*

The bit representations for the type column are:

Decimal	Hex	Status
1	0x1	Exclusive table lock
2	0x2	Shared table lock
3	0x3	Exclusive intent lock (will do page locking on indicated pages)
4	0x4	Shared intent lock
5	0x5	Exclusive page lock
6	0x6	Shared page lock
7	0x7	Update page lock (changes to exclusive if page is actually modified)
256	0x100	Lock is blocking another process
512	0x200	Demand lock

Table B-29: type *control bit in the* syslocks *table*

Indexes

none

Referenced by System Procedures

sp_lock

sysloginroles

***(master* database only)**

Description

sysloginroles contains a row for each instance of a server login possessing a system-defined role. One row is added for each role possessed by each login. For example, if a single server user is granted three roles, three rows are added to *sysloginroles* associated with that user's *suid*.

Column	Datatype	Description
suid	*suid*	Server user ID
srid	*smallint*	Server role ID; one of the following: 0 **sa_role** 1 **sso_role** 2 **oper_role** 4 **navigator_role** 5 **replication_role** 6 **bcpin_labels_role**
status	*smallint*	Reserved

Table B-30: Columns in the sysloginroles *table*

Indexes

Clustered index on *suid*

Referenced by System Procedures

sp_displaylogin, sp_droplogin, sp _locklogin, sp_role

syslogins

(*master* database only)

Description

syslogins contains one row for each valid SQL Server user account. On the SQL Server distribution tape, *syslogins* contains an entry in which the name is "sa", the suid is 1, and the password is null. It also contains an entry named "probe" with an unpublished password. The login "probe" and the user "probe" exist for the Two Phase Commit Probe Process, which uses a challenge and response mechanism to access SQL Server.

Column	Datatype	Description
suid	*smallint*	Server user ID
status	*smallint*	Status of the account. See Table B-32.
accdate	*datetime*	Date *totcpu* and *totio* were last cleared
totcpu	*int*	CPU time accumulated by login
totio	*int*	I/O accumulated by login
spacelimit	*int*	Reserved
timelimit	*int*	Reserved
resultlimit	*int*	Reserved
dbname	*sysname*	Name of database in which to put user when connection established
name	*sysname*	Login name of user
password	*varbinary*	Password of user (encrypted)
language	*varchar(30)*	User's default language
pwdate	*datetime*	Date the password was last changed
audflags	*int*	User's audit settings
fullname	*varchar(30)*	Full name of the user

Table B-31: Columns in the syslogins *table*

The bit representations for the *status* column are:

Decimal	Hex	Status
1	0x1	Password less than 6 characters, or NULL
2	0x2	Account is locked
4	0x4	Password is expired

Table B-32: status *control bits in the* syslogins *table*

Indexes

unique clustered index on *suid*
unique nonclustered index on *name*

Referenced by System Procedures

sp_addalias, sp_addlogin, sp_addremotelogin, sp_adduser, sp_changedbowner, sp_checknames, sp_checkreswords, sp_clearstats, sp_displaylogin, sp_droplogin, sp_helpdb, sp_helpuser, sp_locklogin, sp_modifylogin, sp_reportstats, sp_role

syslogs

(all databases)

Description

syslogs contains the transaction log. It is used by SQL Server for recovery and roll forward, and is not useful to users.

You cannot delete from, insert into, or update *syslogs*. Every data modification operation is logged, so before you can change *syslogs*, the change must be logged. This means that a change operation on *syslogs* adds a row to *syslogs*, which then must be logged, adding another row to *syslogs*, and so on, producing an infinite loop. The loop continues until the database becomes full.

Column	Datatype	Description
xactid	*binary(6)*	Transaction ID
op	*tinyint*	Update operation number

Table B-33: Columns in the syslogs *table*

Indexes

none

sysmessages

(*master* database only)

Description

sysmessages contains one row for each system error or warning that can be returned by SQL Server. SQL Server displays the error description on the user's screen.

Column	Datatype	Description
error	*int*	Unique error number
severity	*smallint*	Severity level of error
dlevel	*smallint*	Reserved for number of descriptive level of this message: terse, short, or long
description	*varchar(25)*	Explanation of error with place holders for parameters
langid	*smallint*	Language, null for us_english
sqlstate	*varchar(5)*	SQLSTATE value for the error

Table B-34: Columns in the sysmessages *table*

Indexes

clustered index on *error, dlevel*
unique nonclustered index on *error, dlevel, langid*

Referenced by System Procedures

sp_configure, sp_dboption, sp_depends, sp_droplanguage, sp_getmessage, sp_help, sp_helpdb, sp_helpdevice, sp_helpremotelogin, sp_remoteoption

sysobjects

(all databases)

Description

sysobjects contains one row for each table, view, stored procedure, log, rule, default, trigger, check constraint, referential constraint, and (in *tempdb* only) temporary object.

Column	Datatype	Description
name	*sysname*	Object name
id	*int*	Object ID
uid	*smallint*	User ID of object owner
type	*char(2)*	One of the following object types: S system table U user table V view L log P procedure R rule D default TR trigger RI referential constraint
userstat	*smallint*	Application-dependent type information (32768 decimal [0x8000 hex] indicates to Data Workbench™ that a procedure is a report)
sysstat	*smallint*	Internal status information (256 decimal [0x100 hex] indicates that table is read-only)
indexdel	*smallint*	Index delete count (incremented if an index is deleted)
schemacnt	*smallint*	Count of changes in schema of a given object (incremented if a rule or default is added)
sysstat2	*smallint*	Additional internal status information. See Table B-36.
crdate	*datetime*	Date object was created
expdate	*datetime*	Reserved
deltrig	*int*	Stored procedure ID of a delete trigger
instrig	*int*	Stored procedure ID of an insert trigger

Table B-35: Columns in the sysobjects *table*

Column	Datatype	Description
updtrig	*int*	Stored procedure ID of an update trigger
seltrig	*int*	Reserved
ckfirst	*int*	ID of first check constraint on the table
cache	*smallint*	Reserved
audflags	*int*	Object's audit settings
objspare	*int*	Spare

Table B-35: Columns in the sysobjects *table (continued)*

The bit representations for the *sysstat2* column are:

Decimal	Hex	Status
1	0x1	Table has referential constraint
2	0x2	Table has foreign key constraint
4	0x4	Table has more than one check constraint
8	0x8	Table has primary key constraint
16	0x10	Chained transaction mode only stored procedure
32	0x20	Any transaction mode stored procedure
64	0x40	Table has IDENTITY field

Table B-36: systat2 *control bits in the* sysobjects *table*

Indexes

unique clustered index on *id*
unique nonclustered index on *name, uid*

Referenced by System Procedures

sp_addmessage, sp_addthreshold, sp_bindefault, sp_bindmsg, sp_bindrule, sp_checknames, sp_checkreswords, sp_column_privileges, sp_columns, sp_commonkey, sp_depends, sp_dropgroup, sp_dropkey, sp_dropsegment, sp_dropthreshold, sp_droptype, sp_dropuser, sp_estspace, sp_fkeys, sp_foreignkey, sp_help, sp_helpconstraint, sp_helpindex, sp_helpjoins, sp_helpkey, sp_helprotect, sp_helpthreshold, sp_indsuspect, sp_modifythreshold, sp_pkeys, sp_placeobject, sp_primarykey, sp_procxmode, sp_recompile, sp_remap, sp_rename, sp_spaceused, sp_sproc_columns, sp_statistics, sp_stored_procedures, sp_table_privileges, sp_tables, sp_unbindefault, sp_unbindmsg, sp_unbindrule

sysprocedures

(all databases)

Description

sysprocedures contains entries for each view, default, rule, trigger, procedure, declarative default, and check constraint. The plan or sequence tree for each object is stored in binary form. If the sequence tree doesn't fit in one entry, it is broken into more than one row. The *sequence* column identifies the sub-rows.

Column	Datatype	Description
type	*smallint*	Object type. See Table B-38.
id	*int*	Object ID
sequence	*smallint*	Sequence number if more than one row is used to describe this object
status	*smallint*	Internal system status
number	*smallint*	Sub-procedure number when the procedure is grouped (0 for non-procedure entries)

Table B-37: Columns in the sysprocedures *table*

The bit representations for the *type* column are:

Decimal	Hex	Status
1	0x1	Entry describes a plan (reserved)
2	0x2	Entry describes a tree

Table B-38: type *control bits in the* sysprocedures *table*

Indexes

unique clustered index on *id, type, sequence, number*

Referenced by System Procedures

sp_bindefault, sp_bindrule, sp_remap, sp_sproc_columns, sp_stored_procedures, sp_unbindefault, sp_unbindrule

sysprocesses

(*master* database only)

Description

sysprocesses contains information about SQL Server processes, but it is not a normal table. Rather, it is built dynamically when queried by a user. No updates to *sysprocesses* are allowed.

Use the **kill** statement to kill a process.

Column	Datatype	Description
spid	*smallint*	Process ID
kpid	*int*	Kernel process ID
enginenum	*int*	Number of engine on which process is being executed
status	*char(12)*	Process ID status, one of: infected background recv sleep send sleep alarm sleep lock sleep sleeping runnable running stopped bad status log suspend
suid	*smallint*	Server user ID of user who issued command
hostname	*char(10)*	Name of host computer
program_name	*char(16)*	Name of front-end module
hostprocess	*char(8)*	Host process ID number
cmd	*char(16)*	Command currently being executed
cpu	*int*	Cumulative cpu time for process in ticks
physical_io	*int*	Number of disk reads and writes for current command
memusage	*int*	Amount of memory allocated to process
blocked	*smallint*	Process ID of blocking process, if any

Table B-39: Columns in the sysprocesses *table*

Column	Datatype	Description
dbid	*smallint*	Database ID
uid	*smallint*	ID of user who executed command
gid	*smallint*	Group ID of user who executed command
tran_name	*varchar*(64)	Name of the active transaction
time_blocked	*int*	Time blocked in seconds
network_pktsz	*int*	Current connection's network packet size

Table B-39: Columns in the sysprocesses *table*

Indexes

none

Referenced by System Procedures

sp_dboption, sp_droplogin, sp_locklogin, sp_role, sp_who

sysprotects

(all databases)

Description

sysprotects contains information on user permissions information—entries for each **grant** and **revoke** statement that has been issued.

Column	Datatype	Description
id	*int*	ID of object to which this permission applies
uid	*smallint*	ID of user or group to which this permission applies
action	*tinyint*	One of the following permissions: **select =** 193 **insert** = 195 **delete** = 196 **update** = 197 **execute** = 224 **references** = 151 **create database** = 203 **create default** = 233 **create procedure** = 222 **create rule** = 236 **create table** = 198 **create view** = 207 **dump database** = 228 **dump transaction** = 235
protecttype	*tinyint*	One of the following values: **grant with grant** = 0 **grant** (permanent) = 1 **revoke** (permanent) = 2
columns	*varbinary(32)*	Bit map of columns to which this **select** or update permission applies. Bit 0 indicates all columns; 1 means permission applies to that column; null means no information.
grantor	*smallint*	User ID of the grantor

Table B-40: Columns in the sysprotects *table*

Indexes

unique clustered index on *id, action, grantor, uid, protecttype*

Referenced by System Procedures

sp_changegroup, sp_dropgroup, sp_dropuser, sp_helprotect, sp_stored_procedures, sp_tables

sysreferences

(all databases)

Description

sysreferences contains one row for each referential integrity constraint declared on a table or column.

Column	Datatype	Description
indexid	*smallint*	ID of the unique index on referenced columns
constrid	*int*	Object ID of the constraint from *sysobjects*
tableid	*int*	Object ID of the referencing table
reftabid	*int*	Object ID of the referenced table
keycnt	*smallint*	The number of columns in the foreign key
status	*smallint*	Reserved
frgndbid	*smallint*	Reserved
pmrydbid	*smallint*	Reserved
spare2	*int*	Reserved
fokey1	*tinyint*	Column ID of the first referencing column
.		
.		
.		
fokey16	*tinyint*	Column ID of the 16th referencing column
refkey1	*tinyint*	Column ID of the first referenced column
.		
.		
.		
refkey16	*tinyint*	Column ID of the 16th referenced column
frgndbname	*varchar(30)*	Name of the database that includes the referencing table (the table with the foreign key). Null if the referencing table is in the current database.
pmrydbname	*varchar(30)*	Name of the database that includes the referenced table (the table with the primary key). Null if the referenced table is in the current database.

Table B-41: Columns in the sysreferences *table*

Indexes

clustered index on *frgndbname, tableid*
unique nonclustered index on *frgndbname, constrid*
nonclustered index on *pmrydbname, reftabid, indexid*

Referenced by System Procedures

sp_fkeys, sp_helpconstraint

sysremotelogins

(*master* database only)

Description

s*ysremotelogins* contains one row for each remote user who is allowed to execute remote procedure calls on this SQL Server.

Column	Datatype	Description
remoteserverid	*smallint*	Identifies the remote server
remoteusername	*varchar*(30)	User's login name on remote server
suid	*smallint*	Local server user ID
status	*smallint*	Bitmap of options

Table B-42: Columns in the sysremotelogins *table*

Indexes

unique clustered index on *remoteserverid, remoteusername*

Referenced by System Procedures

sp_addremotelogin, sp_checknames, sp_checkreswords, sp_dropremotelogin, sp_dropserver, sp_helpremotelogin, sp_remoteoption

sysroles

(all databases)

Description

sysroles maps server role IDs to local role IDs.

Column	Datatype	Description
id	*smallint*	Server role ID (*srid*)
lrid	*smallint*	Local role ID
type	*smallint*	Unused
status	*smallint*	Unused

Table B-43: Columns in the sysroles *table*

Indexes

Unique clustered index on *lrid*

Referenced by System Procedures

None

syssegments

(all databases)

Description

syssegments contains one row for each segment (named collection of disk pieces). The default entries are: segment 0 (*system*) for system tables; segment 2 *(logsegment*) for the transaction log; and segment 1 (*default*) for other objects. Each database has an entry in *sysusages* contain these segments in its maps.

Column	Datatype	Description
segment	*smallint*	Segment number
name	*sysname*	Segment name
status	*int null*	Indicates which segment is default segment

Table B-44: Columns in the syssegments *table*

Indexes

none

Referenced by System Procedures

sp_addsegment, sp_addthreshold, sp_checknames, sp_checkreswords, sp_dropsegment, sp_dropthreshold, sp_dropuser, sp_extendsegment, sp_helpdb, sp_helpindex, sp_helpsegment, sp_helpthreshold, sp_modifythreshold, sp_placeobject

sysservers

(*master* database only)

Description

sysservers contains one row for each remote SQL Server, Backup Server, or Open Server on which this SQL Server can execute remote procedure calls.

Column	Datatype	Description
srvid	*smallint*	ID number (for local use only) of the remote server
srvstatus	*smallint*	Bitmap of options
srvname	*varchar(30)*	Server name
srvnetname	*varchar(32)*	Interfaces file name for the server

Table B-45: Columns in the sysservers *table*

Indexes

unique clustered index on *srvid*
unique nonclustered index on *srvname*

Referenced by System Procedures

sp_addremotelogin, sp_addserver, sp_checknames, sp_checkreswords, sp_configure, sp_dropremotelogin, sp_dropserver, sp_helpremotelogin, sp_helpserver, sp_remoteoption, sp_serveroption

syssrvroles

(*master* database only)

Description

syssrvroles contains a row for each server-wide role.

Column	Datatype	Description
srid	*smallint*	Server role ID
name	*varchar*	Name of the role

Table B-46: Columns in the syssrvroles *table*

Indexes

Unique clustered index on *srid*

Referenced by System Procedures

sp_adduser, sp_changegroup, sp_displaylogin, sp_dropgroup, sp_helpgroup, sp_role

systhresholds

(all databases)

Description

systhresholds contains one row for each threshold defined for the database.

Column	Datatype	Description
segment	*smallint*	Segment number for which free space is being monitored
free_space	*int*	Size of threshold, in 2K pages (4K for Stratus)
status	*smallint*	Bit 1 equals 1 for the logsegment's last-chance threshold, 0 for all other thresholds
proc_name	*varchar(255)*	Name of the procedure that is executed when the number of unused pages on *segment* falls below *free_space*.
suid	*smallint*	The server user ID of the user who added the threshold or modified it most recently
currauth	*varbinary*(255)	A bit mask that indicates which roles were active for *suid* at the time the threshold was added or most recently modified. When the threshold is crossed, *proc_name* executes with this set of roles, less any that have been deactivated since the threshold was added or last modified.

Table B-47: Columns in the systhresholds *table*

Indexes

Unique clustered index on *segment, free_space*

Referenced by System Procedures

sp_addthreshold, sp_dropsegment, sp_dropthreshold, sp_dropuser, sp_helpthreshold, sp_modifythreshold

systypes

(all databases)

Description

systypes contains one row for each system-supplied and user-defined datatype. Domains (defined by rules) and defaults are given, if they exist.

The rows that describe system-supplied datatypes cannot be altered.

Column	Datatype	Description
uid	*smallint*	User ID of datatype creator
usertype	*smallint*	User type ID
variable	*bit*	1 if datatype is variable length; 0 otherwise
allownulls	*bit*	Indicates whether nulls are allowed for this datatype
type	*tinyint*	Physical storage datatype
length	*tinyint*	Physical length of datatype
tdefault	*int*	ID of system procedure that generates default for this datatype
domain	*int*	ID of system procedure that contains integrity checks for this datatype
name	*sysname*	Datatype name
printfmt	*varchar(255)*	Reserved
prec	*tinyint*	Number of significant digits
scale	*tinyint*	Number of digits to the right of the decimal point
ident	*tinyint*	1 if column has the IDENTITY property, 0 if not
hierarchy	*tinyint*	Precedence of the datatype in mixed mode arithmetic

Table B-48: Columns in the systypes *table*

The listing that follows includes the system-supplied datatype *name*, *hierarchy*, *type* (not necessarily unique), and *usertype* (unique). The

datatypes are ordered by *hierarchy*. In mixed mode arithmetic, the datatype with the lowest *hierarchy* takes precedence:

name	hierarchy	type	usertype
floatn	1	109	14
float	2	62	8
datetimn	3	111	15
datetime	4	61	12
real	5	59	23
numericn	6	108	28
numeric	7	63	10
decimaln	8	106	27
decimal	9	55	26
moneyn	10	110	17
money	11	60	11
smallmoney	12	122	21
smalldatetime	13	58	22
intn	14	38	13
int	15	56	7
smallint	16	52	6
tinyint	17	48	5
bit	18	50	16
varchar	19	39	2
sysname	19	39	18
nvarchar	19	39	25
char	20	47	1
nchar	20	47	24
varbinary	21	37	4
timestamp	21	37	80
binary	22	45	3
text	23	35	19
image	24	34	20

Table B-49: Datatype names, hierarchy, types, and usertypes

Indexes

unique clustered index on *name*
unique nonclustered index on *usertype*

Referenced by System Procedures

sp_addtype, sp_bindefault, sp_bindrule, sp_checknames, sp_checkreswords, sp_columns, sp_datatype_info, sp_droptype, sp_dropuser, sp_help, sp_rename, sp_special_columns, sp_sproc_columns,sp_unbindefault, sp_unbindrule

sysusages

(*master* database only)

Description

sysusages contains one row for each disk allocation piece assigned to a database. Each database contains a specified number of database (logical) page numbers. Each disk piece includes the segments on the SQL Server distribution tape, segments 0 and 1.

The **create database** command checks *sysdevices* and *sysusages* to find available disk allocation pieces. One or more contiguous disk allocation pieces is assigned to the database, and the mapping is recorded in *sysusages*.

Column	Datatype	Description
dbid	*smallint*	Database ID
segmap	*int*	Bit map of possible segment assignments
lstart	*int*	First database (logical) page number
size	*int*	Number of contiguous database (logical) pages
vstart	*int*	Starting virtual page number
pad	*smallint*	Unused
unreservedpgs	*int*	Free space not part of an allocated extent

Table B-50: Columns in the sysusages *table*

Indexes

unique clustered index on *dbid, lstar*t
unique nonclustered index on *vstart*

Referenced by System Procedures

sp_addsegment, sp_addthreshold, sp_databases, sp_dropdevice, sp_dropsegment, sp_extendsegment, sp_helpdb, sp_helplog, sp_helpsegment, sp_logdevice, sp_modifythreshold, sp_spaceused

sysusermessages

(all databases)

Description

sysusermessages contains one row for each user-defined message that can be returned by SQL Server.

Column	Datatype	Description
error	*int*	Unique error number. Must be 20000 or above.
uid	*smallint*	User ID of the message creator
description	*varchar(255)*	User-defined message with optional place holders for parameters
langid	*smallint*	Language ID for this message; null for us_english

Table B-51: Columns in the sysusermessages *table*

Indexes

clustered index on *error*
unique nonclustered index on *error*, *langid*

Referenced by System Procedures

sp_addmessage, sp_bindmsg, sp_dropmessage, sp_getmessage, sp_helpconstraint

sysusers

(all databases)

Description

sysusers contains one row for each user allowed in the database, and one row for each group or role.

On the SQL Server distribution tape, *master..sysusers* contains some initial users: "dbo," whose *suid* is 1 and *uid* is 1; "guest," whose *suid* is -1 and *uid* is 2; and "public," whose *suid* is -2 and *uid* is 0. In addition, each role (**sa_role**, **sso_role**, and so on) is listed in *sysusers*, because SQL Server treats roles much like groups.

The user *guest* provides a mechanism for giving users not explicitly listed in *sysusers* access to the database with a restricted set of permissions. The "guest" entry in *master* means that any user with an account on SQL Server (that is, with an entry in *syslogins*) can access *master*.

The user "public" refers to all users. The keyword **public** is used with the **grant** and **revoke** commands to signify that permission is being given to or taken away from all users.

Column	Datatype	Description
suid	*smallint*	Server user ID, copied from *syslogins*.
uid	*smallint*	User ID, unique in this database, used for granting and revoking permissions. User ID 1 is "dbo".
gid	*smallint*	Group ID to which this user belongs. If *uid* = *gid*, this entry defines a group. The group "public" has *suid* = -2; all other groups have *suid*= - *gid*.
name	*sysname*	User or group name, unique in this database
environ	*varchar(255)*	Reserved

Table B-52: Columns in the sysusers *table*

Indexes

unique clustered index on *suid*
unique nonclustered index on *name*
unique nonclustered index on *uid*

Referenced by System Procedures

sp_addalias, sp_addgroup, sp_adduser, sp_changedbowner, sp_changegroup, sp_checknames, sp_checkreswords, sp_column_privileges, sp_depends, sp_dropgroup, sp_droptype, sp_dropuser, sp_helpgroup, sp_helprotect, sp_helpuser, sp_indsuspect, sp_stored_procedures, sp_table_privileges, sp_tables

C The *pubs2* Database

This is the sample database *pubs2*. The names of the 11 tables are *publishers, authors, titles, titleauthor, sales, salesdetail, stores, discounts, roysched, au_pix,* and *blurbs.*

The header for each column lists its datatype (including the user-defined datatypes) and its null / not null status. Defaults, rules, triggers, and indexes are noted where they apply.

Tables in the *pubs2* Database

publishers			
pub_id char(4) not null pub_idrule[a] clust, uniq	pub_name varchar(40) null	city varchar(20) null	state char(2) null
0736	New Age Books	Boston	MA
0877	Binnet & Hardley	Washington	DC
1389	Algodata Infosystems	Berkeley	CA

a. The pub_id rule states that the data must be 1389, 0736, 0877, 1622, or 1756, or must match the pattern 99[0-9][0-9].

authors								
au_id id not null clust, uniq	au_lname varchar(40) not null nonclust	au_fname varchar(20) not null	phone char(12) not null UNKNOWN[a]	address varchar(12) null	city varchar(20) null	state char(2) null	country varchar(12) null	postalcode char(10) null
172-32-1176	White	Johnson	408 496-7223	10932 Bigge Rd.	Menlo Park	CA	USA	94025
213-46-8915	Green	Marjorie	415 986-7020	309 63rd St. #411	Oakland	CA	USA	94618
238-95-7766	Carson	Cheryl	415 548-7723	589 Darwin Ln.	Berkeley	CA	USA	94705
267-41-2394	O'Leary	Michael	408 286-2428	22 Cleveland Av. #14	San Jose	CA	USA	95128
274-80-9391	Straight	Dick	415 834-2919	5420 College Av.	Oakland	CA	USA	94609
341-22-1782	Smith	Meander	913 843-0462	10 Mississippi Dr.	Lawrence	KS	USA	66044
409-56-7008	Bennet	Abraham	415 658-9932	6223 Bateman St.	Berkeley	CA	USA	94705
427-17-2319	Dull	Ann	415 836-7128	3410 Blonde St.	Palo Alto	CA	USA	94301
472-27-2349	Gringlesby	Burt	707 938-6445	PO Box 792	Covelo	CA	USA	95428
486-29-1786	Locksley	Chastity	415 585-4620	18 Broadway Av.	San Francisco	CA	USA	94130
527-72-3246	Greene	Morningstar	615 297-2723	22 Graybar House Rd.	Nashville	TN	USA	37215
648-92-1872	Blotchet-Halls	Reginald	503 745-6402	55 Hillsdale Bl.	Corvallis	OR	USA	97330
672-71-3249	Yokomoto	Akiko	415 935-4228	3 Silver Ct.	Walnut Creek	CA	USA	94595
712-45-1867	del Castillo	Innes	615 996-8275	2286 Cram Pl. #86	Ann Arbor	MI	USA	48105
722-51-5454	DeFrance	Michel	219 547-9982	3 Balding Pl.	Gary	IN	USA	46403
724-08-9931	Stringer	Dirk	415 843-2991	5420 Telegraph Av.	Oakland	CA	USA	94609
724-80-9391	MacFeather	Stearns	415 354-7128	44 Upland Hts.	Oakland	CA	USA	94612
756-30-7391	Karsen	Livia	415 534-9219	5720 McAuley St.	Oakland	CA	USA	94609
807-91-6654	Panteley	Sylvia	301 946-8853	1956 Arlington Pl.	Rockville	MD	USA	20853
846-92-7186	Hunter	Sheryl	415 836-7128	3410 Blonde St.	Palo Alto	CA	USA	94301
893-72-1158	McBadden	Heather	707 448-4982	301 Putnam	Vacaville	CA	USA	95688
899-46-2035	Ringer	Anne	801 826-0752	67 Seventh Av.	Salt Lake City	UT	USA	84152
998-72-3567	Ringer	Albert	801 826-0752	67 Seventh Av.	Salt Lake City	UT	USA	84152

a. The default *UNKNOWN* is inserted if no data is entered.

titles									
title_id tid not null deltitle[3] clust, uniq	title varchar(80) not null nonclust	type char(12) not null UNDECIDED[1]	pub_id char(4) null	price money null	advance money null	total_sales int null	notes varchar(200) null	pubdate datetime not null getdate()[2]	contract bit not null
BU1032	The Busy Executive's Database Guide	business	1389	19.99	5000.00	4095	An overview of available database systems with emphasis on common business applications. Illustrated.	Jun 6, 1986	1
BU1111	Cooking with Computers: Surreptitious Balance Sheets	business	1389	11.95	5000.00	3876	Helpful hints on how to use your electronic resources to the best advantage.	Jun 9, 1988	1
BU2075	You Can Combat Computer Stress!	business	0736	2.99	10125.00	18722	The latest medical and psychological techniques for living with the electronic office. Easy-to-understand explanations.	Jun 30, 1985	1
BU7832	Straight Talk About Computers	business	1389	19.99	5000.00	4095	Annotated analysis of what computers can do for you: a no-hype guide for the critical user.	Jun 22, 1987	1
MC2222	Silicon Valley Gastronomic Treats	mod_cook	0877	19.99	0.00	2032	Favorite recipes for quick, easy, and elegant meals, tried and tested by people who never have time to eat, let alone cook.	Jun 9, 1989	1
MC3021	The Gourmet Microwave	mod_cook	0877	2.99	15000.00	22246	Traditional French gourmet recipes adapted for modern microwave cooking.	Jun 18, 1985	1
PC1035	But Is It User Friendly?	popular_comp	1389	22.95	7000.00	8780	A survey of software for the naive user, focusing on the 'friendliness' of each.	Jun 30, 1986	1
MC3026	The Psychology of Computer Cooking	UNDECIDED	0877	NULL	NULL	NULL	NULL	Jul 24, 1991	0
PC8888	Secrets of Silicon Valley	popular_comp	1389	20.00	8000.00	4095	Muckraking reporting by two courageous women on the world's largest computer hardware and software manufacturers.	Jun 12, 1987	1

1. The default *UNDECIDED* is inserted if no data is entered in the column.
2. The *getdate* function inserts the current date as the default if no data is entered in the column.
3. The *deltitle* trigger prohibits deleting a title if the *title_id* is listed in the *sales* table.

titles

title_id tid not null deltitle[3] clust, uniq	title varchar(80) not null nonclust	type char(12) not null UNDECIDED[1]	pub_id char(4) null	price money null	advance money null	total_sales int null	notes varchar(200) null	pubdate datetime not null getdate()[2]	contract bit not null
PC9999	Net Etiquette	popular_comp	1389	NULL	NULL	NULL	A must-read for computer conferencing debutantes!	Jul 24, 1991	0
PS1372	Computer Phobic and Non-Phobic Individuals: Behavior Variations	psychology	0877	21.59	7000.00	375	A must for the specialist, this book examines the difference between those who hate and fear computers and those who think they are swell.	Oct 21,1990	1
PS2091	Is Anger the Enemy?	psychology	0736	10.95	2275.00	2045	Carefully researched study of the effects of strong emotions on the body. Metabolic charts included.	Jun 15, 1989	1
PS2106	Life Without Fear	psychology	0736	7.00	6000.00	111	New exercise, meditation, and nutritional techniques that can reduce the shock of daily interactions. Popular audience. Sample menus included, exercise video available separately.	Oct 5, 1990	1
PS3333	Prolonged Data Deprivation: Four Case Studies	psychology	0736	19.99	2000.00	4072	What happens when the data runs dry? Searching evaluations of information-shortage effects on heavy users.	Jun 12, 1988	1
PS7777	Emotional Security: A New Algorithm	psychology	0736	7.99	4000.00	3336	Protecting yourself and your loved ones from undue emotional stress in the modern world. Use of computer and nutritional aids emphasized.	Jun 12, 1988	1
TC3218	Onions, Leeks, and Garlic: Cooking Secrets of the Mediterranean	trad_cook	0877	20.95	7000.00	375	Profusely illustrated in color, this makes a wonderful gift book for a cuisine-oriented friend.	Oct 21, 1990	1
TC4203	Fifty Years in Buckingham Palace Kitchens	trad_cook	0877	11.95	4000.00	15096	More anecdotes from the Queen's favorite cook describing life among English royalty. Recipes, techniques, tender vignettes.	Jun 12, 1985	1

1. The default *UNDECIDED* is inserted if no data is entered in the column.
2. The *getdate* function inserts the current date as the default if no data is entered in the column.
3. The *deltitle* trigger prohibits deleting a title if the *title_id* is listed in the *sales* table.

titles									
title_id tid not null deltitle[3] clust, uniq	title varchar(80) not null nonclust	type char(12) not null UNDECIDED[1]	pub_id char(4) null	price money null	advance money null	total_sales int null	notes varchar(200) null	pubdate datetime not null getdate()[2]	contract bit not null
TC7777	Sushi, Anyone?	trad_cook	0877	14.99	8000.00	4095	Detailed instructions on improving your position in life by learning how to make authentic Japanese sushi in your spare time. 5-10% increase in number of friends per recipe reported from beta test.	Jun 12, 1987	1

1. The default *UNDECIDED* is inserted if no data is entered in the column.
2. The *getdate* function inserts the current date as the default if no data is entered in the column.
3. The *deltitle* trigger prohibits deleting a title if the *title_id* is listed in the *sales* table.

titleauthor			
au_id id not null nonclust uniq, clust, composite	title_id tid not null nonclust	au_ord tinyint null	royaltyper int null
172-32-1176	PS3333	1	100
213-46-8915	BU1032	2	40
213-46-8915	BU2075	1	100
238-95-7766	PC1035	1	100
267-41-2394	BU1111	2	40
267-41-2394	TC7777	2	30
274-80-9391	BU7832	1	100
409-56-7008	BU1032	1	60
427-17-2319	PC8888	1	50
472-27-2349	TC7777	3	30
486-29-1786	PC9999	1	100
486-29-1786	PS7777	1	100
648-92-1872	TC4203	1	100
672-71-3249	TC7777	1	40
712-45-1867	MC2222	1	100
722-51-5454	MC3021	1	75
724-80-9391	BU1111	1	60
724-80-9391	PS1372	2	25
756-30-7391	PS1372	1	75
807-91-6654	TC3218	1	100
846-92-7186	PC8888	2	50
899-46-2035	MC3021	2	25
899-46-2035	PS2091	2	50
998-72-3567	PS2091	1	50
998-72-3567	PS2106	1	100

au_pix					
au_id id not null	pic image null	format_type char(11) null	bytesize int null	pixwidth_hor char(14) null	pixwidth_vert char(14) null
409-56-7008	0x0000...	PICT	30220	626	635
486-29-1786	0x59a6...	Sunraster	27931	647	640
648-92-1872	0x59a6...	Sunraster	36974	647	640
672-71-3249	0x000a...	PICT	13487	654	639
899-46-2035	0x4949...	TIF	52023	648	641
998-72-3567	0x4949...	TIF	52336	653	637

The *pic* column contains binary data, which is not reproduced in this table in its entirety. The pictures represented by this data are shown on the next page. Since the *image* data (six pictures, two each in PICT, TIF, and Sun raster file formats) is quite large, you should run the *installpix2* script **only** if you want to use or test the *image* datatype. The *image* data is supplied to show how SYBASE stores *image* data. Sybase does not supply any tools for displaying *image* data: you must use the appropriate screen graphics tools in order to display the images once you have extracted them from the database.

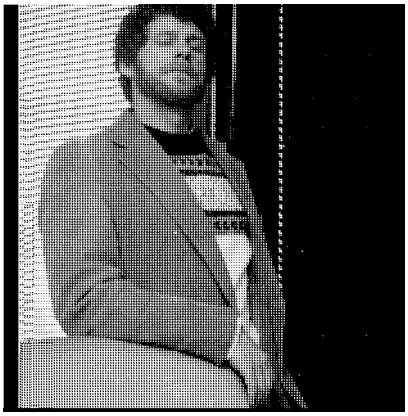
Authors' Portraits from the *au_pix* Table

Akiko Yokomoto 672-71-3249

Chastity Locksley 486-29-1786

Anne Ringer 899-46-2035

Albert Ringer 998-72-3567



Bennet Abraham 409-56-7008

Reginald Blotchet-Halls 648-92-1872

salesdetail				
stor_id char(4) not null nonclust	ord_num varchar(20) not null	title_id tid not null title_idrule nonclust	qty smallint not null	discount float not null
7896	234518	TC3218	75	40.000000
7896	234518	TC7777	75	40.000000
7131	Asoap432	TC3218	50	40.000000
7131	Asoap432	TC7777	80	40.000000
5023	XS-135-DER-432-8J2	TC3218	85	40.000000
8042	91-A-7	PS3333	90	45.000000
8042	91-A-7	TC3218	40	45.000000
8042	91-A-7	PS2106	30	45.000000
8042	91-V-7	PS2106	50	45.000000
8042	55-V-7	PS2106	31	45.000000
8042	91-A-7	MC3021	69	45.000000
5023	BS-345-DSE-860-1F2	PC1035	1000	46.700000
5023	AX-532-FED-452-2Z7	BU2075	500	46.700000
5023	AX-532-FED-452-2Z7	BU1032	200	46.700000
5023	AX-532-FED-452-2Z7	BU7832	150	46.700000
5023	AX-532-FED-452-2Z7	PS7777	125	46.700000
5023	NF-123-ADS-642-9G3	TC7777	1000	46.700000
5023	NF-123-ADS-642-9G3	BU1032	1000	46.700000
5023	NF-123-ADS-642-9G3	PC1035	750	46.700000
7131	Fsoap867	BU1032	200	46.700000
7066	BA52498	BU7832	100	46.700000
7066	BA71224	PS7777	200	46.700000
7066	BA71224	PC1035	300	46.700000
7066	BA71224	TC7777	350	46.700000
5023	ZD-123-DFG-752-9G8	PS2091	1000	46.700000
7067	NB-3.142	PS2091	200	46.700000
7067	NB-3.142	PS7777	250	46.700000
7067	NB-3.142	PS3333	345	46.700000
7067	NB-3.142	BU7832	360	46.700000
5023	XS-135-DER-432-8J2	PS2091	845	46.700000
5023	XS-135-DER-432-8J2	PS7777	581	46.700000
5023	ZZ-999-ZZZ-999-0A0	PS1372	375	46.700000
7067	NB-3.142	BU1111	175	46.700000
5023	XS-135-DER-432-8J2	BU7832	885	46.700000
5023	ZD-123-DFG-752-9G8	BU7832	900	46.700000
5023	AX-532-FED-452-2Z7	TC4203	550	46.700000
7131	Fsoap867	TC4203	350	46.700000
7896	234518	TC4203	275	46.700000
7066	BA71224	TC4203	500	46.700000

salesdetail				
stor_id char(4) not null	ord_num varchar(20) not null	title_id tid not null title_idrule	qty smallint not null	discount float not null
nonclust		nonclust		
7067	NB-3.142	TC4203	512	46.700000
7131	Fsoap867	MC3021	400	46.700000
5023	AX-532-FED-452-2Z7	PC8888	105	46.700000
5023	NF-123-ADS-642-9G3	PC8888	300	46.700000
7066	BA71224	PC8888	350	46.700000
7067	NB-3.142	PC8888	335	46.700000
7131	Asoap432	BU1111	500	46.700000
7896	234518	BU1111	340	46.700000
5023	AX-532-FED-452-2Z7	BU1111	370	46.700000
5023	ZD-123-DFG-752-9G8	PS3333	750	46.700000
8042	13-J-9	BU7832	300	51.700000
8042	13-E-7	BU2075	150	51.700000
8042	13-E-7	BU1032	300	51.700000
8042	13-E-7	PC1035	400	51.700000
8042	91-A-7	PS7777	180	51.700000
8042	13-J-9	TC4203	250	51.700000
8042	13-E-7	TC4203	226	51.700000
8042	13-E-7	MC3021	400	51.700000
8042	91-V-7	BU1111	390	51.700000
5023	AB-872-DEF-732-2Z1	MC3021	5000	50.000000
5023	NF-123-ADS-642-9G3	PC8888	2000	50.000000
5023	NF-123-ADS-642-9G3	BU2075	2000	50.000000
5023	GH-542-NAD-713-9F9	PC1035	2000	50.000000
5023	ZA-000-ASD-324-4D1	PC1035	2000	50.000000
5023	ZA-000-ASD-324-4D1	PS7777	1500	50.000000
5023	ZD-123-DFG-752-9G8	BU2075	3000	50.000000
5023	ZD-123-DFG-752-9G8	TC7777	1500	50.000000
5023	ZS-645-CAT-415-1B2	BU2075	3000	50.000000
5023	ZS-645-CAT-415-1B2	BU2075	3000	50.000000
5023	XS-135-DER-432-8J2	PS3333	2687	50.000000
5023	XS-135-DER-432-8J2	TC7777	1090	50.000000
5023	XS-135-DER-432-8J2	PC1035	2138	50.000000
5023	ZZ-999-ZZZ-999-0A0	MC2222	2032	50.000000
5023	ZZ-999-ZZZ-999-0A0	BU1111	1001	50.000000
5023	ZA-000-ASD-324-4D1	BU1111	1100	50.000000
5023	NF-123-ADS-642-9G3	BU7832	1400	50.000000
5023	BS-345-DSE-860-1F2	TC4203	2700	50.000000
5023	GH-542-NAD-713-9F9	TC4203	2500	50.000000
5023	NF-123-ADS-642-9G3	TC4203	3500	50.000000

salesdetail				
stor_id char(4) not null nonclust	ord_num varchar(20) not null	title_id tid not null title_idrule nonclust	qty smallint not null	discount float not null
5023	BS-345-DSE-860-1F2	MC3021	4500	50.000000
5023	AX-532-FED-452-2Z7	MC3021	1600	50.000000
5023	NF-123-ADS-642-9G3	MC3021	2550	50.000000
5023	ZA-000-ASD-324-4D1	MC3021	3000	50.000000
5023	ZS-645-CAT-415-1B2	MC3021	3200	50.000000
5023	BS-345-DSE-860-1F2	BU2075	2200	50.000000
5023	GH-542-NAD-713-9F9	BU1032	1500	50.000000
5023	ZZ-999-ZZZ-999-0A0	PC8888	1005	50.000000
7896	124152	BU2075	42	50.500000
7896	124152	PC1035	25	50.500000
7131	Asoap132	BU2075	35	50.500000
7067	NB-1.142	PC1035	34	50.500000
7067	NB-1.142	TC4203	53	50.500000
8042	12-F-9	BU2075	30	55.500000
8042	12-F-9	BU1032	94	55.500000
7066	BA27618	BU2075	200	57.200000
7896	124152	TC4203	350	57.200000
7066	BA27618	TC4203	230	57.200000
7066	BA27618	MC3021	200	57.200000
7131	Asoap132	MC3021	137	57.200000
7067	NB-1.142	MC3021	270	57.200000
7067	NB-1.142	BU2075	230	57.200000
7131	Asoap132	BU1032	345	57.200000
7067	NB-1.142	BU1032	136	57.200000
8042	12-F-9	TC4203	300	62.200000
8042	12-F-9	MC3021	270	62.200000
8042	12-F-9	PC1035	133	62.200000
5023	AB-123-DEF-425-1Z3	TC4203	2500	60.500000
5023	AB-123-DEF-425-1Z3	BU2075	4000	60.500000
6380	342157	BU2075	200	57.200000
6380	342157	MC3021	250	57.200000
6380	356921	PS3333	200	46.700000
6380	356921	PS7777	500	46.700000
6380	356921	TC3218	125	46.700000
6380	234518	BU2075	135	46.700000
6380	234518	BU1032	320	46.700000
6380	234518	TC4203	300	46.700000
6380	234518	MC3021	400	46.700000

sales		
stor_id char(4) not null	ord_num varchar(20) not null	date datetime not null
clust, uniq		
5023	AB-123-DEF-425-1Z3	Oct 31 1985
5023	AB-872-DEF-732-2Z1	Nov 6 1985
5023	AX-532-FED-452-2Z7	Dec 1 1990
5023	BS-345-DSE-860-1F2	Dec 12 1986
5023	GH-542-NAD-713-9F9	Mar 15 1987
5023	NF-123-ADS-642-9G3	Jul 18 1987
5023	XS-135-DER-432-8J2	Mar 21 1991
5023	ZA-000-ASD-324-4D1	Jul 27 1988
5023	ZD-123-DFG-752-9G8	Mar 21 1991
5023	ZS-645-CAT-415-1B2	Mar 21 1991
5023	ZZ-999-ZZZ-999-0A0	Mar 21 1991
6380	234518	Sep 30 1987
6380	342157	Dec 13 1985
6380	356921	Feb 17 1991
7066	BA27618	Oct 12 1985
7066	BA52498	Oct 27 1987
7066	BA71224	Aug 5 1988
7067	NB-1.142	Jan 2 1987
7067	NB-3.142	Jun 13 1990
7131	Asoap132	Nov 16 1986
7131	Asoap432	Dec 20 1990
7131	Fsoap867	Sep 8 1987
7896	124152	Aug 14 1986
7896	234518	Feb 14 1991
8042	12-F-9	Jul 13 1986
8042	13-E-7	May 23 1989
8042	13-J-9	Jan 13 1988
8042	55-V-7	Mar 20 1991
8042	91-A-7	Mar 20 1991
8042	91-V-7	Mar 20 1991

stores							
stor_id char(4) not null	stor_name varchar(40) null	stor_address varchar(40) null	city varchar(20) null	state char(2) null	country varchar(12) null	postalcode char(10) null	payterms varchar(12) null
7066	Barnum's	567 Pasadena Ave.	Tustin	CA	USA	92789	Net 30
7067	News & Brews	577 First St.	Los Gatos	CA	USA	96745	Net 30
7131	Doc-U-Mat: Quality Laundry and Books	24-A Avrogado Way	Remulade	WA	USA	98014	Net 60
8042	Bookbeat	679 Carson St.	Portland	OR	USA	89076	Net 30
6380	Eric the Read Books	788 Catamaugus Ave.	Seattle	WA	USA	98056	Net 60
7896	Fricative Bookshop	89 Madison St.	Fremont	CA	USA	90019	Net 60
5023	Thoreau Reading Discount Chain	20435 Walden Expressway	Concord	MA	USA	01776	Net 60

discounts				
discounttype varchar(40) not null	stor_id char(4) null	lowqty smallint null	highqty smallint null	discount float not null
Initial Customer				10.5
Volume Discount		100	1000	6.7
Huge Volume Discount		1001		10
Customer Discount	8042			5

roysched			
title_id tid not null nonclust	lorange int null	hirange int null	royalty int null
BU1032	0	5000	10
BU1032	5001	50000	12
PC1035	0	2000	10
PC1035	2001	3000	12
PC1035	3001	4000	14
PC1035	4001	10000	16
PC1035	10001	50000	18
BU2075	0	1000	10
BU2075	1001	3000	12
BU2075	3001	5000	14
BU2075	5001	7000	16
BU2075	7001	10000	18
BU2075	10001	12000	20
BU2075	12001	14000	22
BU2075	14001	50000	24
PS2091	0	1000	10
PS2091	1001	5000	12
PS2091	5001	10000	14
PS2091	10001	50000	16
PS2106	0	2000	10
PS2106	2001	5000	12
PS2106	5001	10000	14
PS2106	10001	50000	16
MC3021	0	1000	10
MC3021	1001	2000	12
MC3021	2001	4000	14
MC3021	4001	6000	16
MC3021	6001	8000	18
MC3021	8001	10000	20
MC3021	10001	12000	22

roysched			
title_id tid not null nonclust	lorange int null	hirange int null	royalty int null
MC3021	12001	50000	24
TC3218	0	2000	10
TC3218	2001	4000	12
TC3218	4001	6000	14
TC3218	6001	8000	16
TC3218	8001	10000	18
TC3218	10001	12000	20
TC3218	12001	14000	22
TC3218	14001	50000	24
PC8888	0	5000	10
PC8888	5001	10000	12
PC8888	10001	15000	14
PC8888	15001	50000	16
PS7777	0	5000	10
PS7777	5001	50000	12
PS3333	0	5000	10
PS3333	5001	10000	12
PS3333	10001	15000	14
PS3333	15001	50000	16
BU1111	0	4000	10
BU1111	4001	8000	12
BU1111	8001	10000	14
BU1111	12001	16000	16
BU1111	16001	20000	18
BU1111	20001	24000	20
BU1111	24001	28000	22
BU1111	28001	50000	24
MC2222	0	2000	10
MC2222	2001	4000	12
MC2222	4001	8000	14

roysched			
title_id tid not null nonclust	lorange int null	hirange int null	royalty int null
MC2222	8001	12000	16
MC2222	8001	12000	16
MC2222	12001	20000	18
MC2222	20001	50000	20
TC7777	0	5000	10
TC7777	5001	15000	12
TC7777	15001	50000	14
TC4203	0	2000	10
TC4203	2001	8000	12
TC4203	8001	16000	14
TC4203	16001	24000	16
TC4203	24001	32000	18
TC4203	32001	40000	20
TC4203	40001	50000	22
BU7832	0	5000	10
BU7832	5001	10000	12
BU7832	10001	15000	14
BU7832	15001	20000	16
BU7832	20001	25000	18
BU7832	25001	30000	20
BU7832	30001	35000	22
BU7832	35001	50000	24
PS1372	0	10000	10
PS1372	10001	20000	12
PS1372	20001	30000	14
PS1372	30001	40000	16
PS1372	40001	50000	18

blurbs	
au_id id not null	copy text null
486-29-1786	If Chastity Locksley didn't exist, this troubled world would have created her! Not only did she master the mystic secrets of inner strength to conquer adversity when she encountered it in life, but, after "reinventing herself", as she says, by writing "Emotional Security: A New Algorithm" following the devastating loss of her cat Old Algorithm, she also founded Publish or Perish, the page-by-page, day-by-day, write-yourself-to-wellness encounter workshops franchise empire, the better to share her inspiring discoveries with us all. Her "Net Etiquette," a brilliant social treatise in its own right and a fabulous pun, is the only civilized alternative to the gross etiquette often practiced on the public networks.
648-92-1872	A chef's chef and a raconteur's raconteur, Reginald Blotchet-Halls calls London his second home. "Th' palace kitchen's me first 'ome, act'lly!" Blotchet-Halls' astounding ability to delight our palates with palace delights is matched only by his equal skill in satisfying our perpetual hunger for delicious back-stairs gossip by serving up tidbits and entrees literally fit for a king!
998-72-3567	Albert Ringer was born in a trunk to circus parents, but another kind of circus trunk played a more important role in his life years later. He grew up as an itinerant wrestler and roustabout in the reknowned Ringer Brothers and Betty and Bernie's Circus. Once known in the literary world only as Anne Ringer's wrestling brother, he became a writer while recuperating from a near-fatal injury received during a charity benefit bout with a gorilla. "Slingshotting" himself from the ring ropes, Albert flew over the gorilla's head and would have landed head first on the concrete. He was saved from certain death by Nana, an elephant he had befriended as a child, who caught him in her trunk. Nana held him so tightly that three ribs cracked and he turned blue from lack of oxygen. "I was delirious. I had an out-of-body experience! My whole life passed before me eyes. I promised myself 'If I get through this, I'll use my remaining time to share what I learned out there.' I owe it all to Nana!"
899-46-2035	Anne Ringer ran away from the circus as a child. A university creative writing professor and her family took Anne in and raised her as one of their own. In this warm and television-less setting she learned to appreciate the great classics of literature. The stream of aspiring and accomplished writers that flowed constantly through the house confirmed her repudiation of the circus family she'd been born into: "Barbarians!" The steadily growing recognition of her literary work was, to her, vindication. When her brother's brush with death brought them together after many years, she took advantage of life's crazy chance thing and broke the wall of anger that she had constructed to separate them. Together they wrote, "Is Anger the Enemy?" an even greater blockbuster than her other collaborative work, with Michel DeFrance, "The Gourmet Microwave."
672-71-3249	They asked me to write about myself and my book, so here goes: I started a restaurant called "de Gustibus" with two of my friends. We named it that because you really can't discuss taste. We're very popular with young business types because we're young business types ourselves. Whenever we tried to go out to eat in a group we always got into these long tiresome negotiations: "I just ate Italian," or "I ate Greek yesterday," or "I NEVER eat anything that's not organic!" Inefficient. Not what business needs today. So, it came to us that we needed a restaurant we could all go to every day and not eat the same thing twice in a row maybe for a year! We thought, "Hey, why make people choose one kind of restaurant over another, when what they really want is a different kind of food?" At de Gustibus you can eat Italian, Chinese, Japanese, Greek, Russian, Tasmanian, Iranian, and on and on all at the same time. You never have to choose. You can even mix and match! We just pooled our recipes, opened the doors, and never looked back. We're a big hit, what can I say? My recipes in "Sushi, Anyone?" are used at de Gustibus. They satisfy crowds for us every day. They will work for you, too. Period!
409-56-7008	Bennet was the classic too-busy executive. After discovering computer databases he now has the time to run several successful businesses and sit on three major corporate boards. Bennet also donates time to community service organizations. Miraculously, he also finds time to write and market executive-oriented in-depth computer hardware and software reviews. "I'm hyperkinetic, so being dynamic and fast-moving is a piece of cake. But being organized isn't easy for me or for anyone I know. There's just one word for that: 'databases!' Databases can cure you or kill you. If you get the right one, you can be like me. If you get the wrong one, watch out. Read my book!"

Primary and Foreign Keys in *pubs2*

Primary Keys	
Table	Primary Key
titles	title_id
titleauthor	au_id + title_id
authors	au_id
publishers	pub_id
roysched	title_id
sales	stor_id + ord_num
salesdetail	stor_id + ord_num
stores	stor_id
discounts	discounttype + stor_id
au_pix	au_id
blurbs	au_id

Foreign Keys		
Table	Foreign Key	Primary Key Table
titleauthor	title_id au_id	titles authors
roysched	title_id	titles
sales	title_id stor_id	titles stores
salesdetail	title_id stor_id, ord_num	titles sales
titles	pub_id	publishers
discounts	stor_id	stores
au_pix	au_id	authors
blurbs	au_id	authors

Other Objects in *pubs2*

Rules

pub_idrule

```
create rule pub_idrule
as @pub_id in (“1389”, “0736”, “0877”, “1622”,
“1756”)
or @pub_id like “99[0-9][0-9]”
```

title_idrule

```
create rule title_idrule
as
@title_id like “BU[0-9][0-9][0-9][0-9]” or
@title_id like “[MT]C[0-9][0-9][0-9][0-9]” or
@title_id like “P[SC][0-9][0-9][0-9][0-9]” or
@title_id like “[A-Z][A-Z]xxxx” or
@title_id like “[A-Z][A-Z]yyyy”
/*valid values: BU, MC, TC, PS, PC + 4 digits or
**any two uppercase letters followed by x’s or y’s
*/
```

Defaults

typedflt

```
create default typedflt as “UNDECIDED”
```

datedflt

```
create default datedflt as getdate()
```

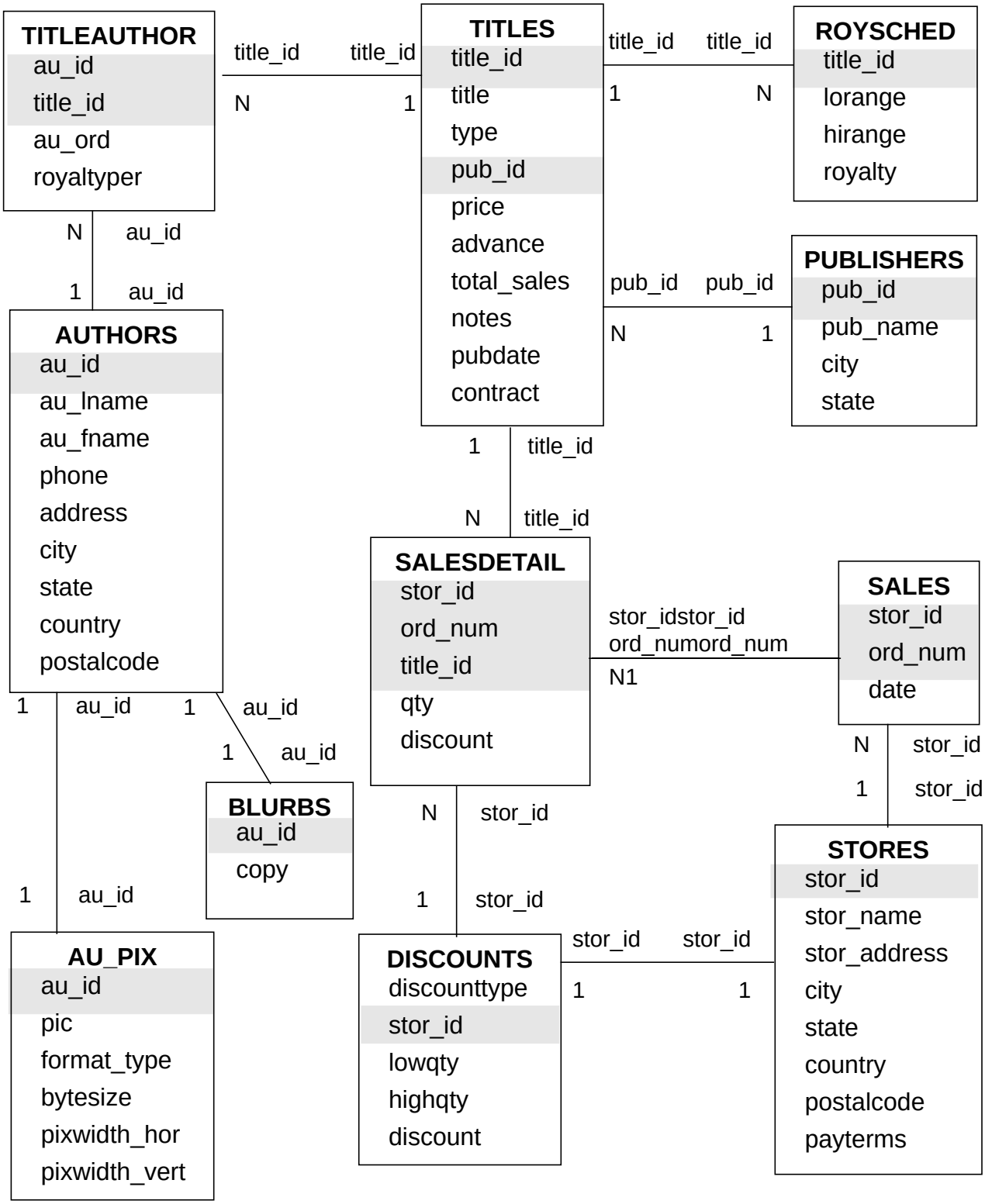
phonedflt

```
create default phonedflt as “UNKNOWN”
```

View

```
create view titleview
as
select title, au_ord, au_lname,
price, total_sales, pub_id
from authors, titles, titleauthor
where authors.au_id = titleauthor.au_id
and titles.title_id = titleauthor.title_id
```

Diagram of the *pubs2* Database



D SQLSTATE Codes and Messages

This appendix describes SQL Server's SQLSTATE status codes and the associated messages. SQLSTATE codes are required for entry level SQL92 compliance. They provide diagnostic information about two types of conditions:

- **Warnings**—conditions that require user notification but are not serious enough to prevent a SQL statement from executing successfully
- **Exceptions**—conditions that prevent a SQL statement from having any effect on the database

Each SQLSTATE code consists of a two-character class followed by a three-character subclass. The class specifies general information about error type; the subclass, more specific information.

SQLSTATE codes are stored in the *sysmessages* system table, along with the messages that display when these conditions are detected. Not all SQL Server error conditions are associated with a SQLSTATE code—only those mandated by SQL92. In some cases, multiple SQL Server error conditions are associated with a single SQLSTATE value.

Warnings

SQL Server currently detects only one SQLSTATE warning condition, which is described in Table D-1:

Message	Value	Description
`Warning - null value eliminated in set function.`	01003	Occurs when using an aggregate function (**avg**, **max**, **min**, **sum**, or **count**) on an expression with a null value.

Table D-1: Warnings

Exceptions

SQL Server detects the following types of exceptions:

- Cardinality violations
- Data exceptions
- Integrity constraint violations
- Invalid cursor states
- Syntax errors and access rule violations
- Transaction rollbacks
- **with check option** violations

Exception conditions are described in Table D-2 through Table D-8. Each class of exceptions appears in its own table. Within each table, conditions are sorted alphabetically by message text.

Cardinality Violations

Cardinality violations occur when a query that should return only a single row returns more than one row to an Embedded SQL application.

Message	Value	Description
`Subquery returned more than 1 value. This is illegal when the subquery follows =, !=, <, <=, >, >=. or when the subquery is used as an expression.`	21000	Occurs when: • A scalar subquery or a row subquery returns more than one row. • A **select into** *parameter_list* query in Embedded SQL returns more than one row.

Table D-2: Cardinality violations

Data Exceptions

Data exceptions occur when an entry:

- Is too long for its datatype
- Contains an illegal escape sequence, or
- Contains other format errors.

Message	Value	Description
`Arithmetic overflow occurred.`	22003	Occurs when: • An exact numeric type would lose precision or scale as a result of an arithmetic operation or **sum** function • An approximate numeric type would lose precision or scale as a result of truncation, rounding, or a **sum** function
`Data exception - string data right truncated.`	22001	Occurs when a *char* or *varchar* column is too short for the data being inserted or updated and non-blank characters must be truncated.
`Divide by zero occurred.`	22012	Occurs when a numeric expression is being evaluated and the value of the divisor is zero.
`Illegal escape character found. there are fewer bytes than necessary to form a valid character.`	22019	Occurs when searching for strings that match a given pattern if the escape sequence does not consist of a single character.
`Invalid pattern string. The character following the escape character must be percent sign, underscore, left square bracket, right square bracket, or the escape character.`	22025	Occurs when searching for strings that match a particular pattern if: • The escape character is not immediately followed by a percent sign, an underscore, or the escape character itself or • The escape character partitions the pattern into substrings whose length is not 1 or 2 characters

Table D-3: Data exceptions

Integrity Constraint Violations

Integrity constraint violations occur when an **insert**, **update**, or **delete** statement violates a **primary key**, **foreign key**, **check**, or **unique** constraint or a unique index.

Message	Value	Description
`Attempt to insert duplicate key row in object` *`object_name`* `with unique index` *`index_name`*	23000	Occurs when a duplicate row is inserted in a table with a unique constraint or index.
`Check constraint violation occurred, dbname =`*`database_name`*`, table name =` *`table_name`*`, constraint name =` *`constraint_name`*	23000	Occurs when an **update** or **delete** would violate a check constraint on a column.
`Dependent foreign key constraint violation in a referential integrity constraint. dbname =` *`database_name`*`, table name =` *`table_name`*`, constraint name =` *`constraint_name`*	23000	Occurs when an **update** or **delete** on a primary key table would violate a foreign key constraint.
`Foreign key constraint violation occurred, dbname =` *`database_name`*`, table name =` *`table_name`*`, constraint name =` *`constraint_name`*	23000	Occurs when an **insert** or **update** on a foreign key table is performed without a matching value in the primary key table.

Table D-4: Integrity constraint violations

Invalid Cursor States

Invalid cursor states occur when:

- A **fetch** uses a cursor that is not currently open,
- An **update where current of** or **delete where current of** affects a cursor row that has been modified or deleted, or
- An **update where current of** or **delete where current of** affects a cursor row that has not been fetched

Message	Value	Description
`Attempt to use cursor` *`cursor_name`* `which is not open. Use the system stored procedure sp_cursorinfo for more information.`	24000	Occurs when an attempt is made to fetch from a cursor that has never been opened or that was closed by a **commit** statement or an implicit or explicit **rollback**. Reopen the cursor and repeat the **fetch**.
`Cursor` *`cursor_name`* `was closed implicitly because the current cursor position was deleted due to an update or a delete. The cursor scan position could not be recovered. This happens for cursors which reference more than one table.`	24000	Occurs when the join column of a multi-table cursor has been deleted or changed. Issue another **fetch** to reposition the cursor.
`The cursor` *`cursor_name`* `had its current scan position deleted because of a DELETE/UPDATE WHERE CURRENT OF or a regular searched DELETE/UPDATE. You must do a new FETCH before doing an UPDATE or DELETE WHERE CURRENT OF.`	24000	Occurs when a user issues an **update/delete where current of** whose current cursor position has been deleted or changed. Issue another **fetch** before retrying the **update/delete where current of**.
`The UPDATE/DELETE WHERE CURRENT OF failed for the cursor` *`cursor_name`* `because it is not positioned on a row.`	24000	Occurs when a user issues an **update/delete where current of** on a cursor that: • Has not yet fetched a row or • Has fetched one or more rows after reaching the end of the result set

Table D-5: Invalid cursor states

Syntax Errors and Access Rule Violations

Syntax errors are generated by SQL statements that contain unterminated comments, implicit datatype conversions not supported by SQL Server, or other incorrect syntax.

Access rule violations are generated when a user tries to access an object that does not exist or one for which he or she does not have the correct permissions.

Message	Value	Description
command permission denied on object *object_name*, database *database_name*, owner *owner_name*.	42000	Occurs when a user tries to access an object for which he or she does not have the proper permissions.
Implicit conversion from datatype '*datatype*' to '*datatype*' is not allowed. Use the CONVERT function to run this query.	42000	Occurs when the user attempts to convert one datatype to another but SQL Server cannot do the conversion implicitly.
Incorrect syntax near *object_name*.	42000	Occurs when incorrect SQL syntax is found near the object specified.
Insert error: column name or number of supplied values does not match table definition.	42000	Occurs during inserts when an invalid column name is used, or when an incorrect number of values has been inserted.
Missing end comment mark '*/'.	42000	Occurs when a comment that begins with the /* opening delimiter does not also have a */ closing delimiter.
object_name not found. Specify owner.objectname or use sp_help to check whether the object exists (sp_help may produce lots of output).	42000	Occurs when a user tries to reference an object that he or she does not own. When referencing an object owned by another user, be sure to qualify the object name with the name of its owner.
The size (*size*) given to the *object_name* exceeds the maximum. The largest size allowed is *size*.	42000	Occurs when: • The total size of all the columns in a table definition exceeds the maximum allowed row size or • The size of a single column or parameter exceeds the maximum allowed for its datatype.

Table D-6: Syntax errors and access rule violations

Transaction Rollbacks

Transaction rollbacks occur when a user has set the **transaction isolation level** to 3, but SQL Server cannot guarantee that concurrent transactions are serializable. This type of exception generally results from system problems such as disk crashes and off-line disks.

Message	Value	Description
`Your server command (process id #process_id) was deadlocked with another process and has been chosen as deadlock victim. Re-run your command.`	40001	Occurs when SQL Server detects that it cannot guarantee the serializability of two or more concurrent transactions.

Table D-7: Transaction rollbacks

With Check Option Violation

This class of exception occurs when data being inserted or updated through a view would not be visible through the view.

Message	Value	Description
`The attempted insert or update failed because the target view was either created WITH CHECK OPTION or spans another view created WITH CHECK OPTION. At least one resultant row from the command would not qualify under the CHECK OPTION constraint.`	44000	Occurs when the view itself, or any view on which it depends, was created with a **with check option** clause.

Table D-8: With check option violations

Index

Index

The index is divided into three sections:

- Symbols

  Indexes each of the symbols used in SYBASE SQL Server documentation.

- Numbers

  Indexes entries which begin numerically.

- Subjects

  Indexes subjects alphabetically.

Page numbers in **bold** are primary references.

Symbols

-- (double hyphen) comments Vol. 1 3-11
- (minus sign)
  arithmetic operator Vol. 1 3-53
  for negative monetary values Vol. 1 3-34
!< (not less than) comparison operator Vol. 1 3-56, Vol. 1 3-80
!= (not equal to) comparison operator Vol. 1 3-56, Vol. 1 3-80
!> (not greater than) comparison operator Vol. 1 3-56, Vol. 1 3-80
" " (quotation marks)
  comparison operators and Vol. 1 3-56
  enclosing constant values Vol. 1 2-37
  enclosing *datetime* values Vol. 1 3-35
  enclosing empty strings Vol. 1 3-59, Vol. 1 3-93
  enclosing punctuation Vol. 1 3-98
  enclosing reserved words Vol. 2 1-90
  enclosing values in Vol. 2 1-7
  in expressions Vol. 1 3-60
  literal specification of Vol. 1 3-60
  single, and **quoted_identifier** Vol. 2 1-98
(pound sign)
  temporary table identifier prefix Vol. 1 1-75, Vol. 1 3-117
$ (dollar sign) in monetary datatypes Vol. 1 3-34
% (percent sign)
  arithmetic operator (modulo) Vol. 1 3-53
  in error messages Vol. 1 1-253
  literal in error messages Vol. 1 1-255
  place holder Vol. 1 1-253
  wildcard character Vol. 1 3-57, Vol. 1 3-106
() (parentheses)
  in expressions Vol. 1 2-19, Vol. 1 3-59
  in SQL statements Vol. 1 xix, Vol. 2 xvii
  in user-defined datatypes Vol. 2 1-40
* (asterisk). *See* Asterisks (*)
*= (asterisk equals) outer join operator Vol. 1 3-82
+ (plus)
  arithmetic operator Vol. 1 3-53
  concatenation operator Vol. 1 2-33
, (comma)
  in monetary values Vol. 1 3-34

in SQL statements Vol. 1 xix, Vol. 2 xvii
in user-defined datatypes Vol. 2 1-40
. (period)
preceding milliseconds Vol. 1 2-20
separator for qualifier names Vol. 1 3-63
.. (dots) in database object names Vol. 1 3-63
... (ellipsis) in SQL statements Vol. 1 xxi, Vol. 2 xix
/ (slash), arithmetic operator (division) Vol. 1 3-53
: (colon) preceding milliseconds Vol. 1 2-20
< (less than) comparison operator Vol. 1 3-56, Vol. 1 3-80
<= (less than or equal to) comparison operator Vol. 1 3-56, Vol. 1 3-80
<> (not equal to) comparison operator Vol. 1 3-56, Vol. 1 3-80
= (equal to) comparison operator Vol. 1 3-56, Vol. 1 3-80
=* (equals asterisk) outer join operator Vol. 1 3-83
> (greater than) comparison operator Vol. 1 3-56, Vol. 1 3-80
>= (greater than or equal to) comparison operator Vol. 1 3-56, Vol. 1 3-80
?? (question marks)
for partial characters Vol. 1 1-262
@ (at symbol)
condition expression argument Vol. 1 1-68
local variable name Vol. 1 1-117 to 1-118
procedure arguments and Vol. 1 1-188
@@ (at symbols), global variable name Vol. 1 3-145
[ ] (square brackets)
character set wildcard Vol. 1 3-57, Vol. 1 3-151
in SQL statements Vol. 1 xix, Vol. 2 xvii
[^] (square brackets and caret) character set wildcard Vol. 1 3-57
\ (backslash), character string continuation with Vol. 1 1-335, Vol. 1 3-60
^ (caret)
wildcard character Vol. 1 3-151
_ (underscore)
character string wildcard Vol. 1 3-57, Vol. 1 3-151
object identifier prefix Vol. 1 3-61
in temporary table names Vol. 1 3-61, Vol. 1 3-116
{} (curly braces) in SQL statements Vol. 1 xix, Vol. 2 xvii

Numerics

“0x” Vol. 1 2-10, Vol. 1 2-16, Vol. 1 3-32, Vol. 1 3-33
21st century numbers Vol. 1 3-34
7-bit ASCII characters, checking with **sp_checknames** Vol. 2 1-83
7-bit terminal, **sp_helpsort** output Vol. 2 1-202
8-bit terminal, **sp_helpsort** output Vol. 2 1-203

A

Abbreviations for date parts Vol. 1 2-20
abort tran on log full database option
setting with **sp_dboption** Vol. 2 1-117
Aborting a query Vol. 1 2-27
abs mathematical function **Vol. 1 2-25**
Absolute value mathematical function Vol. 1 2-25
Accent sensitivity
compute and Vol. 1 1-37
dictionary sort order and Vol. 1 1-250
group by and Vol. 1 1-216
wildcard characters and Vol. 1 3-152
Access. *See* Permissions; Users
Accountability. *See* Roles

Accounting
sp_clearstats procedure Vol. 2 1-100
sp_reportstats procedure Vol. 2 1-254 to 1-255
Accounts, locking and unlocking Vol. 2 1-215
acos mathematical function **Vol. 1 2-25**
add keyword, **alter table** Vol. 1 1-10
Adding
aliases Vol. 1 1-326, Vol. 2 1-9 to 1-10
columns to a table Vol. 1 1-10
dump devices Vol. 2 1-46 to 1-48
group to a database Vol. 2 1-13 to 1-14
an interval to a date Vol. 1 2-21
logins Vol. 2 1-20 to 1-22
messages to *sysusermessages* Vol. 1 1-255, Vol. 2 1-23 to 1-24
mirror device Vol. 1 1-134 to 1-136, Vol. 1 3-47 to 3-51
objects to *tempdb* Vol. 1 1-83
remote logins Vol. 2 1-25 to 1-27
rows to a table or view Vol. 1 1-219 to 1-226
segments Vol. 2 1-28 to 1-30
servers Vol. 2 1-31 to 1-33
summary columns Vol. 1 2-5
summary rows Vol. 1 2-29
timestamp column Vol. 1 3-8
user-defined datatypes Vol. 1 3-40, Vol. 2 1-40 to 1-45
users to a database Vol. 2 1-20 to 1-22, Vol. 2 1-49 to 1-51
users to a group Vol. 2 1-13 to 1-14
users using **sp_adduser** procedure Vol. 2 1-49 to 1-51
Addition operator (**+**) Vol. 1 3-53
Aggregate functions **Vol. 1 2-2** to **2-8**
See also Row aggregates; *individual function names*
cursors and Vol. 1 2-7, Vol. 1 3-20
difference from row aggregates Vol. 1 2-30
group by clause and Vol. 1 1-207, Vol. 1 1-209, Vol. 1 2-3, Vol. 1 2-6
having clause and Vol. 1 1-208, Vol. 1 1-209, Vol. 1 2-2, Vol. 1 2-4
not permitted in **where** clause Vol. 1 1-334
not used on virtual tables B-3
null values and Vol. 1 3-93
scalar aggregates Vol. 1 2-5
vector aggregates Vol. 1 2-5
vector aggregates, **group by** and Vol. 1 1-209
views and Vol. 1 1-108
where clause and Vol. 1 1-334
Aggregate-free expression, grouping by Vol. 1 1-208
Aliases
See also Logins; Users
alternate language Vol. 2 1-261 to 1-262
assigning Vol. 2 1-9 to 1-10
assigning different names compared to Vol. 2 1-49
column Vol. 1 1-208, Vol. 1 1-209
compute clauses allowing Vol. 1 1-34
database ownership transfer and Vol. 2 1-79
dropping Vol. 2 1-131 to 1-132, Vol. 2 1-157
language name B-31, Vol. 2 1-15, Vol. 2 1-261 to 1-262
select Vol. 1 1-283
server Vol. 2 1-31
sp_addalias and Vol. 1 1-326, Vol. 2 1-9
sp_helpuser and Vol. 2 1-209
sysalternates table B-5, Vol. 2 1-9, Vol. 2 1-131
all keyword
aggregate functions and Vol. 1 2-2, Vol. 1 2-3
comparison operators and Vol. 1 3-113
grant Vol. 1 1-196, Vol. 1 1-203
group by Vol. 1 1-207
negated by **having** clause Vol. 1 1-208
revoke Vol. 1 1-269

searching with Vol. 1 3-108
select Vol. 1 1-281, Vol. 1 1-288
subqueries using Vol. 1 3-57, Vol. 1 3-113
union Vol. 1 1-313
where Vol. 1 1-329
Allocation map. *See* Object Allocation Map (OAM)
Allocation report
dbcc indexalloc Vol. 1 1-112
dbcc tablealloc Vol. 1 1-111
Allocation units
sysusages table B-57
allow nulls by default database option
setting with **sp_dboption** Vol. 2 1-117
allow updates configuration variable B-3
setting with **sp_configure** Vol. 2 1-106
allow_dup_row option, **create index** Vol. 1 1-52
alter database command **Vol. 1 1-6** to **1-9**
db_remap and Vol. 2 1-122
alter table command **Vol. 1 1-10** to **1-18**
adding *timestamp* column Vol. 1 3-8
null values and Vol. 1 3-88
Alternate identity. *See* Aliases
Alternate languages. *See* Languages, alternate
and (&) operator
logical Vol. 1 3-58
and keyword
in expressions Vol. 1 3-58
in joins Vol. 1 3-80
range-end Vol. 1 1-330, Vol. 1 3-57
in search conditions Vol. 1 3-107, Vol. 1 3-109
where Vol. 1 1-329
Angles, mathematical functions for Vol. 1 2-25
ANSI compatibility
set options for Vol. 1 1-305
user-defined datatypes and Vol. 2 1-41
ANSI SQL pattern matching Vol. 2 2-2
ansinull option, **set** Vol. 1 1-294
any keyword
in expressions Vol. 1 3-56
searching with Vol. 1 3-108, Vol. 1 3-113
subquery using Vol. 1 3-113
where clause Vol. 1 1-330
Approximate numeric datatypes **Vol. 1 3-28**
arithmetic operations and Vol. 1 3-28
Approximate numeric values, rounding Vol. 1 3-28
APT-SQL keywords list A-3
Arguments
See also Logical expressions
having clause search Vol. 1 1-208
mathematical function Vol. 1 2-24
numbered place holders for Vol. 1 1-253
in **print** command Vol. 1 1-254
stored procedures (parameters) Vol. 1 3-96
string function Vol. 1 2-33
system function Vol. 1 2-40, Vol. 1 2-45
in user-defined error messages Vol. 1 1-257
where clause, number allowed Vol. 1 1-335
arithabort option, **set**
arith_overflow and Vol. 1 1-295, Vol. 1 2-15, Vol. 1 2-27, Vol. 1 3-45
mathematical functions and **arith_overflow** Vol. 1 2-27
mathematical functions and **numeric_truncation** Vol. 1 2-27
numeric_truncation and Vol. 1 2-16
arithignore option, **set**
arith_overflow and Vol. 1 1-295, Vol. 1 2-16
mathematical functions and **arith_overflow** Vol. 1 2-28
Arithmetic expression syntax Vol. 1 3-52

Arithmetic operations
approximate numeric datatypes and Vol. 1 3-28
exact numeric datatypes and Vol. 1 3-27
monetary datatypes and Vol. 1 3-34
Arithmetic operators
in expressions Vol. 1 3-53
subqueries and Vol. 1 3-111
where clause Vol. 1 1-331
Ascending order, **asc** keyword Vol. 1 1-248, Vol. 1 1-286
ASCII characters
checking for with **sp_checknames** Vol. 2 1-83
ascii string function Vol. 1 2-34
asin mathematical function **Vol. 1 2-25**
Asterisks (*)
for overlength numbers Vol. 1 2-38
multiplication operator Vol. 1 3-53
pairs surrounding comments Vol. 1 3-10
select and Vol. 1 1-106
At symbol (@)
condition expression argument Vol. 1 1-68
local variable name Vol. 1 1-117 to 1-118
procedure arguments and Vol. 1 1-188
atan mathematical function **Vol. 1 2-25**
@@char_convert global variable Vol. 1 3-145
@@client_csid global variable Vol. 1 3-145
@@client_csname global variable Vol. 1 3-145
@@connections global variable Vol. 1 3-145
sp_monitor and Vol. 2 1-228
@@cpu_busy global variable Vol. 1 3-146
sp_monitor and Vol. 2 1-228
@@error global variable Vol. 1 1-65, Vol. 1 1-255, Vol. 1 1-259, Vol. 1 3-146
@@identity global variable Vol. 1 1-224, Vol. 1 3-72 to 3-73, Vol. 1 3-146
@@idle global variable Vol. 1 3-146
sp_monitor and Vol. 2 1-228
@@io_busy global variable Vol. 1 3-146
sp_monitor and Vol. 2 1-228
@@isolation global variable Vol. 1 3-133, Vol. 1 3-146
@@langid global variable Vol. 1 1-256, Vol. 1 3-146
@@language global variable Vol. 1 3-146
@@max_connections global variable Vol. 1 3-146
@@maxcharlen global variable Vol. 1 3-146
@@ncharsize global variable Vol. 1 3-147
sp_addtype and Vol. 2 1-42
@@nestlevel global variable Vol. 1 1-66, Vol. 1 1-101, Vol. 1 1-190, Vol. 1 3-147
@@pack_received global variable Vol. 1 3-147
sp_monitor and Vol. 2 1-228
@@pack_sent global variable Vol. 1 3-147
sp_monitor and Vol. 2 1-228
@@packet_errors global variable Vol. 1 3-147
sp_monitor and Vol. 2 1-228
@@procid global variable Vol. 1 3-147
@@rowcount global variable Vol. 1 1-297, Vol. 1 3-147
cursors and Vol. 1 3-21
fetch and Vol. 1 1-194
triggers and Vol. 1 1-99
@@servername global variable Vol. 1 3-147
@@spid global variable Vol. 1 3-147
@@sqlstatus global variable Vol. 1 3-147
cursors and Vol. 1 3-21
fetch and Vol. 1 1-193
@@textsize global variable Vol. 1 1-299, Vol. 1 3-123, Vol. 1 3-148
readtext and Vol. 1 1-262

@@thresh_hysteresis global variable Vol. 1 3-148
@@timeticks global variable Vol. 1 3-148
@@total_errors global variable Vol. 1 3-148, Vol. 2 1-228
@@total_read global variable Vol. 1 3-148
  sp_monitor and Vol. 2 1-228
@@total_write global variable Vol. 1 3-148
  sp_monitor and Vol. 2 1-228
@@tranchained global variable Vol. 1 3-132, Vol. 1 3-148
@@trancount global variable Vol. 1 3-127, Vol. 1 3-148
@@transtate global variable Vol. 1 3-148
@@version global variable Vol. 1 1-254, Vol. 1 3-149
atn2 mathematical function Vol. 1 2-25
Attributes, displaying with **sp_server_info** Vol. 2 2-19 to 2-22
au_pix table, *pubs2* database C-7
audit queue size configuration variable
  setting with **sp_configure** Vol. 2 1-106
Audit trail
  adding comments to Vol. 2 1-11
  comments, adding Vol. 2 1-11
Auditing **Vol. 1 3-3** to **3-5**
  adhoc records option Vol. 2 1-62
  commands Vol. 1 3-3
  databases Vol. 1 3-3, Vol. 2 1-52 to 1-54
  enabling and disabling Vol. 2 1-61
  errors Vol. 1 3-3, Vol. 2 1-62
  global options Vol. 2 1-61
  logins Vol. 1 3-3, Vol. 2 1-61, Vol. 2 1-62
  logouts Vol. 2 1-61
  options, setting Vol. 1 3-4
  privileged commands, use of Vol. 2 1-62
  queue, size of Vol. 1 3-4
  records (comments) and Vol. 2 1-11
  remote procedure calls Vol. 1 3-3, Vol. 2 1-62
  role toggling Vol. 2 1-62
  server boots Vol. 2 1-61
  setting defaults Vol. 2 1-58
  stored procedures Vol. 2 1-65 to 1-67
  sybsecurity database Vol. 1 3-3
  sysauditoptions table Vol. 1 3-3, B-6
  sysaudits table Vol. 1 3-3, B-8
  table access Vol. 2 1-55, Vol. 2 1-58
  triggers Vol. 2 1-65 to 1-67
  users Vol. 1 3-3, Vol. 2 1-55 to 1-57
  users' commands Vol. 2 1-55
  view access Vol. 2 1-55, Vol. 2 1-58
Author *blurbs* table, *pubs2* database C-17
Authority. *See* Permissions
Authorizations. *See* Permissions
authors table, *pubs2* database C-2
auto identity database option
  setting with **sp_dboption** Vol. 2 1-117
Automatic operations
  checkpoints Vol. 1 1-25
  conversion of values Vol. 1 1-83
  timestamp Vol. 1 3-39
  triggers Vol. 1 1-94
avg aggregate function **Vol. 1 2-3**
  as row aggregate Vol. 1 2-29

B

Backslash (\) for character string continuation Vol. 1 1-335, Vol. 1 3-60
Backup Server
  multiple Vol. 2 1-32
  sp_volchanged Vol. 2 1-279
Backups
  See also Dump, database; Dump, transaction log; Load, database; Load, transaction log
  disk mirroring and Vol. 1 1-135, Vol. 1 1-144, Vol. 1 3-50
  disk remirroring and Vol. 1 1-140
  loading Vol. 1 1-230 to 1-237, Vol. 1 1-238 to 1-245
  master database Vol. 1 1-8

transaction log Vol. 1 1-238 to 1-245
Base 10 logarithm function Vol. 1 2-26
Base date Vol. 1 3-38
Base tables. *See* Tables
Batch processing
create default command Vol. 1 1-47
execute Vol. 1 1-187, Vol. 1 1-190
go command Vol. 1 3-6
queries **Vol. 1 3-6** to **3-7**
return status Vol. 1 1-266 to 1-268
bcp (bulk copy utility)
fast version Vol. 2 1-119
IDENTITY columns and Vol. 1 3-73
begin transaction command **Vol. 1 1-21** to **1-22**
commit and Vol. 1 1-28
rollback to Vol. 1 1-275
begin...end commands **Vol. 1 1-19** to **1-20**
if...else and Vol. 1 1-217
triggers and Vol. 1 1-94
between keyword Vol. 1 3-57
check constraint using Vol. 1 1-91
search conditions Vol. 1 3-107
where Vol. 1 1-330
Binary datatypes **Vol. 1 3-32** to **3-33**
"0x" prefix Vol. 1 1-46, Vol. 1 1-68, Vol. 1 3-32
bitwise operations on Vol. 1 3-55
platform dependence and Vol. 1 3-33
trailing zeroes in Vol. 1 3-32
Binary expressions Vol. 1 xxii, Vol. 2 xix
Binary operation, **union** Vol. 1 1-314
Binary sort order
of character sets Vol. 1 1-250, Vol. 2 1-203
Binding
See also **sp_bindefault** system procedure; **sp_bindrule** system procedure
defaults Vol. 1 1-46 to 1-47, Vol. 2 1-68 to 1-71
new default Vol. 1 1-148
rules Vol. 1 1-68, Vol. 1 1-70, Vol. 2 1-75 to 1-78
unbinding and Vol. 2 1-271
user messages to constraints Vol. 2 1-72 to 1-74
bit datatype **Vol. 1 3-39**
Bitwise operators
binary representation by Vol. 1 3-55
in expressions Vol. 1 3-54 to 3-55
subqueries and Vol. 1 3-111
truth tables Vol. 1 3-54
where clause Vol. 1 1-331
Blanks
See also Spaces, character
appended Vol. 1 3-59
character datatypes and Vol. 1 3-29 to 3-32
character datatypes and **insert** Vol. 1 1-221
expressions with trailing Vol. 1 3-56, Vol. 1 3-59
ignored trailing Vol. 1 3-56
removing leading, with **ltrim** function Vol. 1 2-34
removing trailing using **rtrim** function Vol. 1 2-35
in system procedure parameter values Vol. 2 1-7
update and Vol. 1 1-320
using **null** or Vol. 1 3-88
Blocking process Vol. 1 1-227, B-43, Vol. 2 1-286
Blocks, database device Vol. 1 1-131
blurbs table, *pubs2* database C-17
Boolean expressions
select statements in Vol. 1 1-218
syntax Vol. 1 3-52
Brackets and caret wildcard ([^]) Vol. 1 3-106
Brackets wildcard ([]) Vol. 1 3-106
Branching
to label, unconditional Vol. 1 1-195
break command **Vol. 1 1-23** to **1-24**
while loop Vol. 1 1-24, Vol. 1 1-336 to 1-337
Browse mode **Vol. 1 3-8** to **3-9**
cursor declarations and Vol. 1 3-8

select Vol. 1 1-287
and *timestamp* datatype Vol. 1 2-43
Built-in functions Vol. 1 2-1
See also individual function names
date Vol. 1 2-19 to 2-23
image functions Vol. 1 2-48 to 2-50
mathematical **Vol. 1 2-24** to **2-28**
string Vol. 1 2-33 to 2-39
system Vol. 1 2-40 to 2-47
text functions Vol. 1 2-48 to 2-50
Bulk copying. *See* **bcp** (bulk copy utility)
by row aggregate subgroup Vol. 1 1-30, Vol. 1 2-29
Bytes
See also Size
per row Vol. 1 1-15, Vol. 1 1-82
bytes option, **readtext** Vol. 1 1-261

C

Calling procedure
nesting and Vol. 1 1-63, Vol. 1 1-100
return parameter Vol. 1 1-59, Vol. 1 3-98
Canceling a command at **rowcount** Vol. 1 1-298
See also **rollback** command
Canceling an update Vol. 1 1-52
cascade option, **revoke** Vol. 1 1-271, Vol. 1 1-273
Cascading changes (triggers) Vol. 1 1-96
Case sensitivity Vol. 1 xx
in comparison expressions Vol. 1 3-56, Vol. 1 3-152 to 3-153
compute and Vol. 1 1-36
group by and Vol. 1 1-215
of identifier names Vol. 1 3-61, Vol. 1 3-62
in SQL Vol. 2 xviii
sort order and Vol. 1 1-250
of utility commands Vol. 1 3-62
Catalog stored procedures Vol. 2 2-1 to 2-36
list of Vol. 2 2-1
syntax Vol. 2 2-2
ceiling mathematical function **Vol. 1 2-25**
Chain of pages, *text* or *image* data Vol. 1 3-120
chained option, **set** Vol. 1 1-295
Chained transaction mode Vol. 1 3-131
commit and Vol. 1 1-28
delete and Vol. 1 1-127
fetch and Vol. 1 1-193
insert and Vol. 1 1-222
open and Vol. 1 1-246
sp_procxmode and Vol. 2 1-238
update and Vol. 1 1-319
Changes, canceling. *See* **rollback** command
Changing
See also Updating
alternate language alias Vol. 2 1-261
configuration variables Vol. 1 1-264
current database owner Vol. 2 1-79 to 1-80
database options Vol. 2 1-115 to 1-121
syslogs table, dangers of B-3
system tables B-3
user's group Vol. 2 1-81 to 1-82
user's password Vol. 2 1-230 to 1-232
char datatype **Vol. 1 3-29** to **3-30**
in expressions Vol. 1 3-59
row sort order and Vol. 1 1-251
char string function Vol. 1 2-34
@@char_convert global variable Vol. 1 3-145
char_convert option, **set** Vol. 1 1-295
char_length string function Vol. 1 2-34
Character datatypes Vol. 1 3-29
avoiding "NULL" in Vol. 1 3-93
Character expressions, blanks or spaces in Vol. 1 3-29 to 3-32
Character set conversion
between client and server Vol. 1 1-295
set char_convert Vol. 1 1-295
Character sets
changing names of Vol. 2 1-94, Vol. 2 1-97

checking with **sp_checknames** Vol. 2 1-83
checking with **sp_checkreswords** Vol. 2 1-89
fix_text upgrade after change in Vol. 1 1-113
multibyte Vol. 1 3-65, Vol. 2 1-203
object identifiers and Vol. 1 3-65
sp_helpsort display of Vol. 2 1-202
syscharsets system table B-11
Character strings
continuation with (\) Vol. 1 3-60
empty Vol. 1 1-221, Vol. 1 3-59
expression syntax Vol. 1 3-52
specifying quotes within Vol. 1 3-60
truncation Vol. 1 1-221, Vol. 1 1-299
wildcards in Vol. 1 3-57
Characters
See also Spaces, character
"0x" Vol. 1 1-68, Vol. 1 2-10, Vol. 1 2-16, Vol. 1 3-32, Vol. 1 3-33
not converted with **char_convert** Vol. 1 1-295
stuff function for deleting Vol. 1 2-38
wildcard Vol. 1 3-150 to 3-155
Character-type expressions Vol. 1 xxii, Vol. 2 xix
Chargeback accounting
sp_clearstats procedure Vol. 2 1-100 to 1-101
sp_reportstats procedure Vol. 2 1-254 to 1-255
charindex string function Vol. 1 2-34
chars or **characters** option, **readtext** Vol. 1 1-261
Check constraints
binding user messages to Vol. 2 1-72
displaying the text of Vol. 2 1-204
insert and Vol. 1 1-221
renaming Vol. 2 1-248 to 1-250
sysconstraints table B-18
system tables entries for B-40 to B-41, B-42
check option
alter table Vol. 1 1-13
create table Vol. 1 1-79
checkalloc option, **dbcc** Vol. 1 1-111
checkcatalog option, **dbcc** Vol. 1 1-112
checkdb option, **dbcc** Vol. 1 1-111
Checker, consistency. *See* **dbcc** (Database Consistency Checker)
Checking passwords. *See* Passwords; **sp_remoteoption** system procedure
checkpoint command **Vol. 1 1-25** to **1-26**
sp_dboption and Vol. 2 1-117
Checkpoint process Vol. 1 1-25 to 1-26
See also Recovery; Savepoints
checktable option, **dbcc** Vol. 1 1-110 to 1-111
Clearing
accounting statistics Vol. 2 1-100 to 1-101
Client
character set conversion Vol. 1 1-295
cursors Vol. 1 3-17
host computer name Vol. 1 2-42
@@client_csid global variable Vol. 1 3-145
@@client_csname global variable Vol. 1 3-145
close command **Vol. 1 1-27**
close on endtran option, **set** Vol. 1 1-296
Closing cursors Vol. 1 1-27
clustered constraint
alter table Vol. 1 1-12
create table Vol. 1 1-78
Clustered indexes
See also Indexes
create index clustered option for Vol. 1 1-50 to 1-51
migration of tables to Vol. 1 1-55, Vol. 1 1-84
number of total pages used Vol. 1 2-43
pages allocated to Vol. 1 2-46, B-26
used_pgs system function and Vol. 1 2-43
cntrltype option
disk init Vol. 1 1-131

disk reinit Vol. 1 1-138
Code numbers, ODBC datatype Vol. 2 2-3
Codes
datatype Vol. 2 2-3, Vol. 2 2-12
soundex Vol. 1 2-35
col_length system function Vol. 1 2-41, Vol. 1 2-46
col_name system function Vol. 1 2-41
Collating sequence. *See* Sort order
Collision of database creation requests Vol. 1 1-42
Column data. *See* Datatypes
Column identifiers. *See* Identifiers
Column name Vol. 1 2-41
grouping by Vol. 1 1-208, Vol. 1 1-209
in parentheses Vol. 1 2-29
as qualifier Vol. 1 3-63
specifying in **select** Vol. 1 1-282
union result set Vol. 1 1-315
Column pairs. *See* Joins; Keys
Columns
adding to tables Vol. 1 1-10
binding defaults to Vol. 2 1-68 to 1-71
binding rules to Vol. 2 1-75 to 1-78
changing names of Vol. 2 1-92
checking with **sp_checknames** Vol. 2 1-83
common key Vol. 2 1-102 to 1-104
comparing and concatenating Vol. 1 3-79 to 3-84
creating indexes on Vol. 1 1-50 to 1-57
defaults bound to Vol. 1 1-221
defaults for Vol. 1 1-46 to 1-48
finding dependencies Vol. 2 1-92
gaps in IDENTITY values Vol. 1 3-75 to 3-77
grant Vol. 1 1-197
headings Vol. 1 1-208, Vol. 1 1-209, Vol. 1 1-282, Vol. 1 1-290
identifying Vol. 1 3-63
IDENTITY Vol. 1 3-67 to 3-78
information from **sp_columns** Vol. 2 2-8 to 2-9
joins and Vol. 1 3-80, Vol. 2 1-186
length definition Vol. 1 2-41, Vol. 1 2-46, Vol. 1 3-89
length of Vol. 1 2-41
list and **insert** Vol. 1 1-219
naming Vol. 1 1-221
null values and default Vol. 1 1-48, Vol. 1 1-70, Vol. 1 3-92
numeric, and row aggregates Vol. 1 2-29
order by Vol. 1 1-286
per table Vol. 1 1-15
permissions on Vol. 1 1-197, Vol. 1 1-270, Vol. 2 2-4 to 2-7
primary key Vol. 2 1-235, Vol. 2 2-17 to 2-18
renaming Vol. 2 1-248 to 1-250
reserved B-3
rules Vol. 2 1-75 to 1-78
rules bound to Vol. 1 1-221
rules conflict with definitions of Vol. 1 1-70, Vol. 1 3-89
sizes of (list) Vol. 1 3-25 to 3-26
sp_foreignkey Vol. 2 1-166
sp_sproc_columns information on Vol. 2 2-26 to 2-27
system-generated Vol. 1 3-67
unbinding defaults from Vol. 2 1-271 to 1-273
unbinding rules with **sp_unbindrule** Vol. 2 1-276 to 1-278
union of Vol. 1 1-315
variable-length Vol. 1 3-89, B-13
variable-length, and sort order Vol. 1 1-250
views and Vol. 1 1-103
Columns padding. *See* Padding, data
Comma (,)
in monetary values Vol. 1 3-34
in SQL statements Vol. 1 xix, Vol. 2 xvii
in user-defined datatypes Vol. 2 1-40
Command execution delay. *See* **waitfor** command

Command permissions **Vol. 1 1-200** to **1-202**
See also Object permissions; Permissions
grant all Vol. 1 1-203
grant assignment of Vol. 1 1-196 to 1-206
levels Vol. 1 1-199
revoking Vol. 1 1-270
Commands
auditing Vol. 1 3-3
display syntax of Vol. 2 1-265 to 1-267
not allowed in user-defined transactions Vol. 1 3-129
order sensitive Vol. 1 1-202, Vol. 1 1-273
roles Vol. 1 3-101
rowcount range for Vol. 1 1-298
statistics io for Vol. 1 1-299
statistics time information on Vol. 1 1-299
Transact-SQL, summary table Vol. 1 1-1 to 1-5
Comments Vol. 1 3-10 to 3-11
adding to audit trail Vol. 2 1-11
as control-of-flow language Vol. 1 3-12
commit command **Vol. 1 1-28** to **1-29**
begin transaction and Vol. 1 1-21, Vol. 1 1-28
rollback and Vol. 1 1-28, Vol. 1 1-275, Vol. 1 1-279
commit work command. *See* **commit** command
Common keys Vol. 1 1-82
See also Foreign keys; Joins; Primary keys
sp_commonkey Vol. 2 1-102 to 1-104
sp_dropkey procedure Vol. 2 1-137
sp_helpjoins and Vol. 2 1-186
sp_helpkey and Vol. 2 1-188
syskeys table B-29
Comparing values
datatype conversion for Vol. 1 1-335
for joins Vol. 1 3-80
for sort order Vol. 1 1-250 to 1-251
in **where** clause Vol. 1 1-335
Comparison
of date/time datatypes Vol. 1 3-56
difference string function Vol. 1 2-34, Vol. 1 2-39
null-valued operands Vol. 1 1-294
of *timestamp* values Vol. 1 2-43, Vol. 1 3-9
Comparison operators
See also Relational expressions
in expressions Vol. 1 3-56
symbols Vol. 1 3-56
where clause Vol. 1 1-330
Compatibility, data
create default and Vol. 1 1-47
of rule to column datatype Vol. 1 1-69
Compiling
sp_recompile and Vol. 2 1-240 to 1-241
time (**statistics time**) Vol. 1 1-299
without execution (**noexec**) Vol. 1 1-297
Composite indexes Vol. 1 1-50, Vol. 1 1-51, Vol. 1 1-56
compute clause **Vol. 1 1-30** to **1-38**
order by and Vol. 1 1-249, Vol. 1 1-286
select Vol. 1 1-286
using row aggregates Vol. 1 2-6
without **by** Vol. 1 1-35
Concatenation
of rows with matching values Vol. 1 3-79 to 3-84
using **+** string function Vol. 1 2-33
Conceptual (logical) tables Vol. 1 1-96, Vol. 1 1-98
Configuration structure B-19
Configuration variables Vol. 1 1-264
sp_configure procedure Vol. 2 1-105 to 1-110
system tables values B-16, B-19
Connections
transactions and Vol. 1 3-128
@@connections global variable Vol. 1 3-145

sp_monitor and Vol. 2 1-228
Consistency check. *See* **dbcc** (Database Consistency Checker)
Constants Vol. 1 xxi, Vol. 2 xix
in expressions Vol. 1 3-59
identification number Vol. 1 1-227
return parameters in place of Vol. 1 1-189
in string functions Vol. 1 2-33, Vol. 1 2-37
constraint keyword
alter table Vol. 1 1-11
create table Vol. 1 1-77
Constraints
binding user messages to Vol. 2 1-72
changing table Vol. 1 1-10
create table Vol. 1 1-84
cross-database Vol. 1 1-90, Vol. 1 1-157
displaying the text of Vol. 2 1-204
error messages Vol. 1 1-86
information about Vol. 2 1-175
referential integrity Vol. 1 1-88
renaming Vol. 2 1-248 to 1-250
sysconstraints table B-18
sysreferences table B-47
system tables entries for B-40 to B-41
unbinding messages with **sp_unbindmsg** Vol. 2 1-274 to 1-275
unique Vol. 1 1-87
contiguous option (OpenVMS)
disk init Vol. 1 1-131
disk mirror Vol. 1 1-134, Vol. 1 3-48
Contiguous sequence of string values Vol. 1 2-39
Continuation lines, character string Vol. 1 1-335, Vol. 1 3-60
continue command **Vol. 1 1-39** to **1-40**
while loop Vol. 1 1-336 to 1-337
Controller number, device Vol. 2 1-181
Control-of-flow language **Vol. 1 3-12** to **3-13**
begin...end and Vol. 1 1-19
create procedure and Vol. 1 1-60
keywords table Vol. 1 3-12
Conventions
See also Syntax
identifier name Vol. 1 3-63
multiple-line comments Vol. 1 3-10
Transact-SQL syntax Vol. 1 xix to xxi, Vol. 2 xvii to xix
used in manuals Vol. 1 xix, Vol. 2 xvi to xix
Conversion
See also Character set conversion
automatic values Vol. 1 1-83, Vol. 1 3-44
between character sets Vol. 1 3-65
character value to ASCII code Vol. 1 2-34
datatypes Vol. 1 1-83, Vol. 1 2-9 to 2-18, Vol. 1 3-44 to 3-45, Vol. 1 3-89
of dates used with **like** Vol. 1 1-332, Vol. 1 3-38
of degrees to radians Vol. 1 2-26
implicit Vol. 1 3-44, Vol. 1 3-59
integer value to character value Vol. 1 2-34
of lower to higher datatypes Vol. 1 3-59
of lower to upper case, **upper** string function Vol. 1 2-36
mathematical functions, **degrees** Vol. 1 2-25
null values and automatic Vol. 1 1-83, Vol. 1 3-44
of radians to degrees Vol. 1 2-25
for string concatenation Vol. 1 3-55
styles for dates Vol. 1 2-10
of upper to lower case, **lower** string function Vol. 1 2-34
where clause and datatype Vol. 1 1-335
convert function Vol. 1 2-9 to 2-18
for non-character expressions Vol. 1 3-55
text values Vol. 1 3-123
Copying
the *model* database Vol. 1 1-42

with **create database** Vol. 1 1-41 to 1-44
Correlated subqueries Vol. 1 3-115
See also Subqueries
Corrupt indexes. *See* **reindex** option, **dbcc**
cos mathematical function **Vol. 1 2-25**
cot mathematical function **Vol. 1 2-25**
count aggregate function **Vol. 1 2-3**
as row aggregate Vol. 1 2-29
count(*) aggregate function **Vol. 1 2-3**
including null values Vol. 1 3-93
Counters, **while** loop. *See* **while** loop
CPU usage
sp_monitor information Vol. 2 1-228
@@cpu_busy global variable Vol. 1 3-146
sp_monitor and Vol. 2 1-228
create database command **Vol. 1 1-41** to **1-45**
disk init and Vol. 1 1-132
log on option Vol. 1 1-42
log on option compared to **sp_logdevice** Vol. 2 1-217
permission Vol. 1 1-203
create default command **Vol. 1 1-46** to **1-49**
batches and Vol. 1 1-47
create index command **Vol. 1 1-50** to **1-57**
insert and Vol. 1 1-221
sp_extendsegment and Vol. 2 1-163
create procedure command **Vol. 1 1-58** to **1-67**
See also Stored procedures
null values and Vol. 1 3-91
order of parameters in Vol. 1 1-188, Vol. 1 1-189
select * in Vol. 1 1-63
create rule command **Vol. 1 1-68** to **1-71**
create schema command **Vol. 1 1-72** to **1-74**
create table command **Vol. 1 1-75** to **1-93**
column order and Vol. 1 1-250
null values and Vol. 1 1-77, Vol. 1 3-88 to 3-93
sp_extendsegment and Vol. 2 1-163
create trigger command **Vol. 1 1-94** to **1-102**
create view command **Vol. 1 1-103** to **1-109**
Creating
new tables Vol. 1 1-282
triggers Vol. 1 1-94 to 1-102
Curly braces ({}) in SQL statements Vol. 1 xix, Vol. 2 xvii
Currency symbol Vol. 1 3-34
Current configuration values. *See* Configuration variables
Current database
space used by Vol. 2 1-263 to 1-264
Current date Vol. 1 2-19
Current locks, **sp_lock** system procedure Vol. 1 1-229, Vol. 2 1-212
Current processes. *See* Processes (Server tasks)
Current user
suser_id system function Vol. 1 2-43
suser_name system function Vol. 1 2-43
Cursor result set Vol. 1 1-122, Vol. 1 3-14
datatypes and Vol. 1 1-192
returning rows Vol. 1 1-192
cursor rows option, **set** Vol. 1 1-296
Cursors **Vol. 1 3-14** to **3-24**
aggregate functions and Vol. 1 2-7
client Vol. 1 3-17
closing Vol. 1 1-27
compute clause and Vol. 1 1-34
cursor scans Vol. 1 1-122, Vol. 1 3-19
datatype compatibility Vol. 1 1-192
deallocating Vol. 1 1-116
declaring Vol. 1 3-14 to 3-21
defining Vol. 1 1-119
delete and Vol. 1 3-19
deleting rows Vol. 1 1-127
execute Vol. 1 3-17
fetch Vol. 1 1-192
for browse and Vol. 1 3-8
grant and Vol. 1 1-202
group by and Vol. 1 1-209
Halloween problem Vol. 1 1-123, Vol. 1 3-21
indexes Vol. 1 3-19 to 3-21

information about Vol. 2 1-111
in joins Vol. 1 3-84
language Vol. 1 3-18
locking Vol. 1 3-22 to 3-24
opening Vol. 1 1-246
order by and Vol. 1 1-249, Vol. 1 3-19
position Vol. 1 3-14
read-only Vol. 1 1-122
regions Vol. 1 3-18
scope Vol. 1 1-121, Vol. 1 3-18
select and Vol. 1 1-289
server Vol. 1 3-17
subqueries and Vol. 1 3-115
transactions and Vol. 1 3-137 to 3-139
types of Vol. 1 3-17
updatable Vol. 1 1-122, Vol. 1 3-22
update and Vol. 1 1-320, Vol. 1 3-19
curunreservedpgs system function **Vol. 1 2-41**
Custom datatypes. *See* User-defined datatypes
Cyrillic characters Vol. 1 3-65

D

Damaged database, removing and repairing Vol. 1 1-112
Data definition
transactions and Vol. 1 3-128
Data dependency. *See* Dependencies, database object
Data dictionary. *See* System tables
Data integrity Vol. 1 1-221
See also Referential integrity constraints
dbcc utility Vol. 1 1-110
Data padding. *See* Padding, data
data_pgs system function Vol. 1 2-41, Vol. 1 2-46
Database design, logical relationships in Vol. 2 1-102, Vol. 2 1-137, Vol. 2 1-166
Database devices
alter database and Vol. 1 1-6
defaulton or **defaultoff** status Vol. 2 1-127 to 1-128
dropping Vol. 2 1-133 to 1-134
dropping segments from Vol. 2 1-148 to 1-150
last device reference for Vol. 2 1-150
listing of Vol. 2 1-180
new database Vol. 1 1-41
sp_helpdevice system procedure Vol. 2 1-180
status Vol. 2 1-127
sysdevices table B-23
system table entries for B-23
transaction logs on separate Vol. 1 1-135, Vol. 1 1-141, Vol. 1 3-47
Database dump. *See* Dump, database; Dump devices
Database files. *See* Files
Database object owners
See also Database Owners; Ownership
identifiers and Vol. 1 3-64
sp_depends system procedure and Vol. 2 1-124
Database objects
See also individual object names
adding to *tempdb* Vol. 1 1-83
auditing Vol. 2 1-58 to 1-60
binding defaults to Vol. 2 1-68 to 1-71
binding rules to Vol. 2 1-75
dependencies of B-22, Vol. 2 1-124 to 1-126
dependencies of, and recompilation Vol. 2 1-249
display text of Vol. 2 1-204
finding Vol. 2 1-125, Vol. 2 1-173
ID number (**object_id**) Vol. 1 2-42
identifier names Vol. 1 3-61
listings of Vol. 2 1-171
permissions on Vol. 1 1-201, Vol. 2 1-195
pubs2 database C-19
remapping Vol. 2 1-242 to 1-244
renaming Vol. 2 1-248 to 1-250
select_list Vol. 1 1-282

sp_help and Vol. 2 1-171, Vol. 2 1-174
sp_tables list of Vol. 2 2-35 to 2-36
space used by Vol. 2 1-263 to 1-264
sysobjects table B-40 to B-41
user-defined datatypes as Vol. 1 3-40
Database options Vol. 2 1-117 to 1-120
See also Individual option names
displaying with **sp_helpdb** Vol. 2 1-117
sp_dboption and Vol. 2 1-115 to 1-121
Database Owners
See also Database object owners; Permissions
adding users Vol. 2 1-49
changing Vol. 2 1-79
changing the login of Vol. 2 1-79
dbo use only database option Vol. 2 1-117
information on Vol. 2 1-208 to 1-209
name as qualifier Vol. 1 3-63, Vol. 1 3-64
objects and identifiers Vol. 1 3-64
permissions granted by Vol. 1 1-196
stored procedures Vol. 1 1-65
use of **setuser** Vol. 1 1-199
user ID number 1 Vol. 1 2-46
database size configuration variable
in *sysconfigures* Vol. 1 1-43
Databases
See also Database objects
auditing Vol. 1 3-3, Vol. 2 1-52 to 1-54
changing name Vol. 2 1-93
checking with **sp_checknames** Vol. 2 1-83
creating with separate log segment Vol. 1 1-181
dropping segments from Vol. 2 1-148 to 1-150
dump database command Vol. 1 1-162 to 1-173
ID number, **db_id** function Vol. 1 2-41
listing of Vol. 2 1-178
listing with **sp_databases** Vol. 2 2-10
load database command Vol. 1 1-230 to 1-237
load transaction command Vol. 1 1-238 to 1-245
number of Server Vol. 1 1-42
options Vol. 2 1-115 to 1-121
removing and repairing damaged Vol. 1 1-112
renaming Vol. 2 1-251 to 1-253
selecting Vol. 1 1-326
storage extension Vol. 1 1-6
system tables entries for B-20
use command Vol. 1 1-326
Databases, system. *See master* database; *model* database; *sybsystemprocs* database; *tempdb* database
datalength system function Vol. 1 2-41, Vol. 1 2-46
dataserver utility command
disk mirror and Vol. 1 1-136
disk remirror and Vol. 1 1-141
Datatype conversions
bit information Vol. 1 2-17
character information Vol. 1 2-13, Vol. 1 2-14
convert function Vol. 1 2-9 to 2-18
date / time information Vol. 1 2-15
domain errors Vol. 1 2-16
hexadecimal-like information Vol. 1 2-16
hextoint function Vol. 1 2-9, Vol. 1 2-17
image information Vol. 1 2-17
implicit Vol. 1 2-11 to 2-12
inttohex function Vol. 1 2-9, Vol. 1 2-17
money information Vol. 1 2-14
numeric information Vol. 1 2-14, Vol. 1 2-15
overflow errors Vol. 1 2-15
rounding during Vol. 1 2-14
scale errors Vol. 1 2-16
Datatype precedence. *See* Precedence
Datatypes **Vol. 1 3-25** to **3-46**
See also User-defined datatypes; *individual datatype names*
approximate numeric Vol. 1 3-28
binary Vol. 1 3-32 to 3-33

binding defaults to Vol. 2 1-68 to 1-71
character Vol. 1 3-29
codes Vol. 2 2-3, Vol. 2 2-12
comparison in **union** operations Vol. 1 1-315
compatibility of column and default Vol. 1 1-47
converting Vol. 1 1-83, Vol. 1 2-9 to 2-18, Vol. 1 3-44 to 3-45, Vol. 1 3-89
cursor result set and Vol. 1 1-192
date and time **Vol. 1 3-34** to **3-38**
datetime values comparison Vol. 1 3-56
decimal Vol. 1 3-27 to 3-28
declaring Vol. 1 3-41
dropping user-defined Vol. 2 1-155
exact numeric Vol. 1 3-27 to 3-28
extended Vol. 2 2-3
help on Vol. 1 3-41
hierarchy of Vol. 1 3-43, B-55
hierarchy of user-defined Vol. 2 1-42
integer Vol. 1 3-27 to 3-28
invalid in **group by** and **having** clauses Vol. 1 1-209
joins Vol. 1 3-80
lists of B-55
local variables and Vol. 1 1-117
mixed, arithmetic operations on Vol. 1 3-53
monetary Vol. 1 3-34
new Vol. 2 1-40 to 1-45
null columns and Vol. 1 3-44
numeric Vol. 1 3-27 to 3-29
ODBC Vol. 2 2-3
physical Vol. 2 1-40
pubs2 database C-20
sp_datatype_info information on Vol. 2 2-12 to 2-13
sp_help information on Vol. 2 1-171 to 1-174
summaries of Vol. 1 3-25 to 3-26
system Vol. 1 3-25, B-55
systypes table B-55 to B-56
trailing zeroes in Vol. 1 3-32
unbinding defaults from Vol. 2 1-271 to 1-273
unbinding rules with **sp_unbindrule** Vol. 2 1-276 to 1-278
user-defined Vol. 2 1-40 to 1-45
user-defined in temporary tables Vol. 1 3-118
Datatypes, custom. *See* User-defined datatypes
Date functions **Vol. 1 2-19** to **2-23**
See also individual function names
Date parts Vol. 1 2-20
abbreviation names and values Vol. 1 2-20
entering Vol. 1 3-35
order of Vol. 1 1-296, Vol. 2 1-15
dateadd function **Vol. 1 2-21**, Vol. 1 2-22
datediff function **Vol. 1 2-20** to **2-21**, Vol. 1 2-22
datefirst option, **set** Vol. 1 1-296, Vol. 1 2-22
dateformat option, **set** Vol. 1 1-296
datename function **Vol. 1 2-20**, Vol. 1 2-22
datepart function **Vol. 1 2-20**, Vol. 1 2-22
Dates
comparing Vol. 1 3-56
datatypes Vol. 1 3-34 to 3-38
display formats Vol. 1 1-296, Vol. 1 3-35
display formats, **waitfor** command Vol. 1 1-328
earliest allowed Vol. 1 2-22, Vol. 1 3-34
entry formats Vol. 1 3-35
like and Vol. 1 3-38
pre-1753 datatypes for Vol. 1 2-22
recognition of Vol. 1 3-35
searching for Vol. 1 3-38
datetime datatype **Vol. 1 3-34** to **3-38**
See also **set** command
comparison of Vol. 1 3-56
conversion Vol. 1 3-38
date functions and Vol. 1 2-20
dateformat setting Vol. 1 3-37
like and Vol. 1 3-38
set dateformat Vol. 1 3-37

values and comparisons Vol. 1 3-38
day date part Vol. 1 2-20
Day values
alternate language Vol. 2 1-15
date style Vol. 1 2-10
syslanguages table B-31
dayofyear date part abbreviation and values Vol. 1 2-20
db_id system function Vol. 1 2-41
db_name system function Vol. 1 2-42
dbcc (Database Consistency Checker) **Vol. 1 1-110** to **1-115**
See also individual **dbcc** *options*
readtext and Vol. 1 1-263
scripts and **sp_checkreswords** Vol. 2 1-91
space allocation and Vol. 2 1-233
DB-Library programs
browse mode Vol. 1 1-287
changing identifier names and Vol. 2 1-91
dbmoretext Vol. 1 3-122
dbwritetext Vol. 1 3-122
dbwritetext and **dbmoretext, writetext** compared to Vol. 1 1-339
overflow errors Vol. 1 2-7, Vol. 1 2-31, Vol. 1 2-32
prepare transaction Vol. 1 1-252
set options for Vol. 1 1-297, Vol. 1 1-303
transactions and Vol. 1 3-141
waitfor mirrorexit and Vol. 1 1-328
dbmoretext DB-Library function Vol. 1 3-122
dbo use only database option
setting with **sp_dboption** Vol. 2 1-117
dbrepair option, **dbcc** Vol. 1 1-112
dbwritetext DB-Library function Vol. 1 3-122
dd. *See* **day** date part
ddl in tran database option
setting with **sp_dboption** Vol. 2 1-118
Deactivation of disk mirroring Vol. 1 1-143
deallocate cursor command **Vol. 1 1-116**
Deallocating cursors Vol. 1 1-116
Debugging aids
triggers and Vol. 1 1-101
Decimal data Vol. 1 3-27 to 3-28
Decimal numbers
round function and Vol. 1 2-26
str function, representation of Vol. 1 2-36, Vol. 1 2-37
declare command **Vol. 1 1-117** to **1-118**
local variables and Vol. 1 3-143
declare cursor command **Vol. 1 1-119** to **1-124**
Declaring
cursors Vol. 1 3-14 to 3-21
datatypes Vol. 1 3-41
Default database
See also sysdevices table
information Vol. 2 1-181
modifying Vol. 2 1-220
default keyword
alter database Vol. 1 1-6
alter table Vol. 1 1-11
create table Vol. 1 1-76
default language configuration variable Vol. 2 1-20
default segment
alter database Vol. 1 1-9
dropping Vol. 2 1-149
mapping Vol. 2 1-29
defaulton | defaultoff option, **sp_diskdefault** Vol. 2 1-127
Defaults
in batch queries Vol. 1 3-7
bound to columns Vol. 1 1-221
changing names of Vol. 2 1-92
checking name with **sp_checkreswords** Vol. 2 1-88
column values Vol. 1 1-11
conflictions with rules Vol. 1 1-47
create default command **Vol. 1 1-46** to **1-49**
database device Vol. 2 1-127
date format Vol. 1 3-35
date, Server Vol. 1 3-35

definitions and **create default** Vol. 1 1-46 to 1-48
displaying the text of Vol. 2 1-204
dropping Vol. 1 1-148
language Vol. 2 1-20, Vol. 2 1-21
language, and date parts Vol. 1 2-22
modifying language Vol. 2 1-220
procedure parameter Vol. 1 1-59, Vol. 1 3-96
pubs2 database C-19
remapping Vol. 2 1-242 to 1-244
removing user-defined Vol. 1 1-148
renaming Vol. 2 1-248 to 1-250
rules and Vol. 1 1-70
set command options Vol. 1 1-302
sp_bindefault procedure for Vol. 2 1-68 to 1-71
system tables entries for B-15, B-40 to B-41, B-42
unbinding Vol. 2 1-271 to 1-273
defncopy utility command Vol. 2 1-90
degrees mathematical function **Vol. 1 2-25**
Degrees, conversion to radians Vol. 1 2-26
Delayed execution (**waitfor**) Vol. 1 1-327
delete command **Vol. 1 1-125** to **1-129**
auditing use of Vol. 2 1-58
cursors and Vol. 1 3-19
text row Vol. 1 3-122
triggers and Vol. 1 1-97
truncate table compared to Vol. 1 1-311
deleted table
triggers and Vol. 1 1-96, Vol. 1 1-98
Deleting
See also Dropping
files Vol. 2 1-133
Demand locks Vol. 2 1-213
Dependencies, database object
changing names of Vol. 2 1-90
recompilation and Vol. 2 1-249
sp_depends system procedure Vol. 1 1-83, Vol. 2 1-124 to 1-126
sysdepends table B-22
Descending order (**desc** keyword) Vol. 1 1-248, Vol. 1 1-286
detail option, **sp_helpconstraint** Vol. 2 1-175
Device failure
dump transaction command after Vol. 1 1-178, Vol. 1 1-180
Device fragments
number of Vol. 1 1-8, Vol. 1 1-42
sp_helpdb report on Vol. 2 1-178
Device initialization. *See* Initializing
Devices
See also sysdevices table
changing names of Vol. 2 1-94, Vol. 2 1-96
disk mirroring to Vol. 1 1-134 to 1-145, Vol. 1 3-47 to 3-51
information on log Vol. 2 1-192
numbering Vol. 1 1-130, Vol. 1 1-138
secondary Vol. 1 1-135, Vol. 1 3-48
system tables entries for B-23
Diagram, *pubs2* database C-20
Dictionary sort order Vol. 1 1-250
difference string function Vol. 1 2-34, Vol. 1 2-39
Direct updates to system tables B-3, Vol. 2 1-94, Vol. 2 1-106
Dirty pages
updating Vol. 1 1-25 to 1-26
Dirty reads Vol. 1 3-132
Disabling mirroring. *See* Disk mirroring
discounts table, *pubs2* database C-13
Disk allocation pieces B-57
Disk controllers Vol. 1 1-131, Vol. 1 1-138
Disk devices
adding Vol. 2 1-46 to 1-48
sysdevices table B-23
disk init command **Vol. 1 1-130** to **1-133**
master database backup after Vol. 1 1-132
disk mirror command **Vol. 1 1-134** to **1-136**, Vol. 1 3-47 to 3-48
Disk mirroring **Vol. 1 1-134** to **1-145**, **Vol. 1 3-47** to **3-51**

dump database command and Vol. 1 1-172
dump transaction command and Vol. 1 1-186
load database command and Vol. 1 1-236
load transaction command and Vol. 1 1-244
sp_who report on Vol. 2 1-286
unmirroring and Vol. 1 1-143
waitfor mirrorexit Vol. 1 1-327
disk option, **sp_addumpdevice** Vol. 2 1-46
disk refit command **Vol. 1 1-137**
create database and Vol. 1 1-43
disk reinit command **Vol. 1 1-138** to **1-139**
See also **disk init** command
disk remirror command **Vol. 1 1-140** to **1-142**, Vol. 1 3-50
See also Disk mirroring
disk unmirror command **Vol. 1 1-143** to **1-145**, Vol. 1 3-49 to 3-50
See also Disk mirroring
dismount | nodismount option
dump database Vol. 1 1-164
dump transaction Vol. 1 1-177
Display
auditing information Vol. 2 1-56
character sets Vol. 2 1-202
create procedure statement text Vol. 1 1-65
database options Vol. 2 1-115 to 1-121
format for dates Vol. 1 3-35
procedures for information Vol. 1 1-60
setting for command-affected rows Vol. 1 1-297
syntax of modules Vol. 2 1-265
text of database objects Vol. 2 1-204
trigger text Vol. 1 1-97
distinct keyword
aggregate functions and Vol. 1 2-2, Vol. 1 2-3
create view Vol. 1 1-103, Vol. 1 1-108
cursors and Vol. 1 3-20
select Vol. 1 1-281, Vol. 1 1-288
select, null values and Vol. 1 3-94
Dividing tables into groups. *See* **group by**
Division operator (*/*) Vol. 1 3-53
Dollar sign ($) in monetary datatypes Vol. 1 3-34
Domain rules Vol. 1 1-221
create rule command Vol. 1 1-68
mathematical functions errors in Vol. 1 2-27
violations Vol. 1 1-221
Dots (..) for omitted name elements Vol. 1 3-63
double precision datatype **Vol. 1 3-28**
Double-byte characters Vol. 1 3-153
Double-precision floating point values Vol. 1 3-28
Doubling quotes
in character strings Vol. 1 3-30
in expressions Vol. 1 3-60
drop commands
auditing use of Vol. 2 1-52
drop database command **Vol. 1 1-146** to **1-147**
damaged databases and Vol. 1 1-112
drop default command **Vol. 1 1-148** to **1-149**
drop index command **Vol. 1 1-150** to **1-151**
drop keyword, **alter table** Vol. 1 1-14
drop logins option, **sp_dropserver** Vol. 2 1-151
drop procedure command **Vol. 1 1-152** to **1-153**
grouped procedures and Vol. 1 1-152, Vol. 1 1-187
drop rule command **Vol. 1 1-154** to **1-155**
drop table command **Vol. 1 1-156** to **1-158**
drop trigger command **Vol. 1 1-159**
drop view command **Vol. 1 1-160** to **1-161**
dropmessages option, **sp_droplanguage** Vol. 2 1-140
Dropping
See also Deleting
aliased user Vol. 2 1-131 to 1-132

character with **stuff** function Vol. 1 2-38
columns from a table Vol. 1 1-15
corrupt indexes Vol. 1 1-113
cursor rows Vol. 1 3-19
damaged database Vol. 1 1-112
database devices Vol. 2 1-133 to 1-134
databases Vol. 1 1-146 to 1-147
dbcc dbrepair database Vol. 1 1-112
default after unbinding it Vol. 1 1-47, Vol. 1 1-148
group from a database Vol. 2 1-135 to 1-136
grouped procedures Vol. 1 1-58
leading or trailing blanks Vol. 1 2-34
remote logins Vol. 2 1-146 to 1-147
rows from a table Vol. 1 1-125 to 1-129, Vol. 1 1-156
rows from a table using **truncate table** Vol. 1 1-311
segment from a database Vol. 2 1-148 to 1-150
servers Vol. 2 1-151 to 1-152
tables Vol. 1 1-156 to 1-158
tables with triggers Vol. 1 1-98
triggers Vol. 1 1-98
user from a database Vol. 2 1-157 to 1-158
user from a group Vol. 2 1-81
user-defined datatype Vol. 2 1-155 to 1-156
user-defined default Vol. 1 1-47, Vol. 1 1-148
user-defined messages Vol. 2 1-144 to 1-145
user-defined rule Vol. 1 1-154

dump database command **Vol. 1 1-162** to **1-173**
See also Dump, database
after using **create database** Vol. 1 1-43
after using **disk init** Vol. 1 1-132
after using **dump transaction with no_log** Vol. 1 1-175
Backup Server, remote Vol. 1 1-163
blocksize Vol. 1 1-163
dismounting tapes Vol. 1 1-164
dump devices Vol. 1 1-163, Vol. 1 1-169
dump striping Vol. 1 1-164
expiration date Vol. 1 1-164
file name Vol. 1 1-165
initializing / appending Vol. 1 1-164
master database and Vol. 1 1-168
message destination Vol. 1 1-165
messages Vol. 1 1-171, Vol. 1 1-184
new databases and Vol. 1 1-168
restrictions Vol. 1 1-167
rewinding tapes after Vol. 1 1-164
select into and Vol. 1 1-291
tape capacity Vol. 1 1-163
tape density Vol. 1 1-163
volume name Vol. 1 1-163

Dump devices
See also Database devices; Log device
adding Vol. 2 1-46 to 1-48
dropping Vol. 2 1-133 to 1-134
dump database and Vol. 1 1-163
dump transaction and Vol. 1 1-176
listing Vol. 2 1-180
naming Vol. 1 1-163, Vol. 1 1-176, Vol. 1 1-182 to 1-183
number required Vol. 1 1-235
permission and ownership problems Vol. 2 1-47
sysdevices table and B-23
system tables entries for B-23

dump transaction command **Vol. 1 1-174** to **1-186**
See also Dump, transaction log
See also **dump database** command; Dump devices
after using **disk init** Vol. 1 1-132
appending dumps Vol. 1 1-177
Backup Server, remote Vol. 1 1-183
dismounting tapes Vol. 1 1-177
dump devices Vol. 1 1-176
dump striping Vol. 1 1-176, Vol. 1 1-177

expiration date Vol. 1 1-177
file name Vol. 1 1-177, Vol. 1 1-183 to 1-184
initializing tape Vol. 1 1-177
insufficient log space Vol. 1 1-181
message destination Vol. 1 1-178
rewinding tapes after Vol. 1 1-177
select into/bulkcopy and Vol. 1 1-180
sp_logdevice and Vol. 2 1-217
tape capacity Vol. 1 1-176
trunc log on chkpt and Vol. 1 1-180
volume name Vol. 1 1-176, Vol. 1 1-184
with no_log option Vol. 1 1-181
with no_truncate option Vol. 1 1-178, Vol. 1 1-180
with truncate_only option Vol. 1 1-180 to 1-181
Dump, database
across networks Vol. 1 1-168
appending to volume Vol. 1 1-171 to 1-172
Backup Server and Vol. 1 1-169
commands used for Vol. 1 1-167, Vol. 1 1-180
cross-platform not supported Vol. 1 1-167
file name Vol. 1 1-165
interrupted Vol. 2 1-122
loading Vol. 1 1-44, Vol. 1 1-230 to 1-237
master database Vol. 1 1-169
model database Vol. 1 1-169
new databases and Vol. 1 1-168
overwriting Vol. 1 1-164, Vol. 1 1-171 to 1-172
permissions problems Vol. 1 1-168
remote Vol. 1 1-170
scheduling Vol. 1 1-167 to 1-168
successive Vol. 1 1-171, Vol. 1 1-184
system databases Vol. 1 1-168
thresholds and Vol. 1 1-168
volume changes Vol. 1 1-171
Dump, transaction log
across networks Vol. 1 1-182
appending to volume Vol. 1 1-185
command used for Vol. 1 1-180
file name Vol. 1 1-177, Vol. 1 1-183 to 1-184
initializing volume Vol. 1 1-185
loading Vol. 1 1-238 to 1-245
permissions problems Vol. 1 1-180
remote Vol. 1 1-183, Vol. 1 1-184
scheduling Vol. 1 1-182
system databases Vol. 1 1-168
thresholds and Vol. 1 1-182
dumpvolume option
dump database Vol. 2 1-279
dup_in_subquery option, **set** Vol. 1 1-296, Vol. 1 1-302
Duplicate key errors, user transaction Vol. 1 3-141
Duplicate rows
text or *image* Vol. 1 3-123
Duplication
avoidance and **create index** Vol. 1 1-50, Vol. 1 1-52
joins without row Vol. 1 3-82
null values considered as Vol. 1 3-94
of space for a new database Vol. 1 1-44
of a table with no data Vol. 1 1-290
Duplication of text. *See* **replicate** string function
dw. *See* **weekday** date part
dy. *See* **dayofyear** date part
Dynamic configuration variables Vol. 1 1-264
allow updates B-3
Dynamic dumps Vol. 1 1-168, Vol. 1 1-182

E

e or E exponent datatype Vol. 1 3-29
8-bit terminal, **sp_helpsort** output Vol. 2 1-203

Ellipsis (...) in SQL statements Vol. 1 xxi, Vol. 2 xix
else keyword. *See* **if...else** conditions
Embedded spaces. *See* Spaces
Embedding join operations Vol. 1 3-79
Empty string (" ") or (' ')
  not evaluated as null Vol. 1 3-93
  as a single space Vol. 1 1-221, Vol. 1 3-32, Vol. 1 3-59
  updating an Vol. 1 1-319
Enclosing quotes in expressions Vol. 1 3-60
Engines
  sysengines table **B-25**
  system tables entries for B-2, B-25
English language, U.S. B-31
Equal to. *See* Comparison operators
Equijoins Vol. 1 3-81
@@error global variable Vol. 1 1-65, Vol. 1 1-255, Vol. 1 1-259, Vol. 1 3-146
Error handling
  in character set conversion Vol. 1 1-296
  dbcc and Vol. 1 1-114
  domain or range Vol. 1 2-27
  triggers and Vol. 1 1-101
Error messages
  See also SQLSTATE codes
  Backup Server Vol. 2 1-283
  on or off setting (for **char_convert**) Vol. 1 1-296
  printing user-defined Vol. 1 1-255
  system tables entries for B-39
  user-defined Vol. 1 1-256 to 1-260
  user-defined transactions and Vol. 1 3-128
errorexit keyword
  waitfor Vol. 1 1-327
Errors
  See also Error messages
  allocation Vol. 1 1-112
  arithmetic overflow Vol. 1 2-15
  auditing Vol. 1 3-3, Vol. 2 1-62
  convert function Vol. 1 2-13 to 2-16
  datatypes Vol. 1 3-45
  divide-by-zero Vol. 1 2-15
  domain Vol. 1 2-16
  duplicate key Vol. 1 3-141
  number of Vol. 2 1-228
  number of packet Vol. 1 3-147
  numbers for user-defined Vol. 1 1-256
  return status values Vol. 1 1-267
  scale Vol. 1 2-16
  in stored procedures Vol. 1 3-139
  trapping mathematical Vol. 1 2-27
  in user-defined transactions Vol. 1 3-141
Escape characters Vol. 1 3-153
  wildcard characters and Vol. 1 3-154
escape keyword Vol. 1 3-153 to 3-154
  in expressions Vol. 1 3-58
  where Vol. 1 1-331
European characters in object identifiers Vol. 1 3-65
Evaluation order Vol. 1 1-314
Exact numeric datatypes **Vol. 1 3-27** to **3-28**
  arithmetic operations and Vol. 1 3-27
Exception report, **dbcc tablealloc** Vol. 1 1-111
Exclusive locks Vol. 2 1-213
execute command **Vol. 1 1-187** to **1-191**
  create procedure and Vol. 1 1-63
Execute cursors Vol. 1 3-17
Execution delay. *See* **waitfor** command
exists keyword
  in expressions Vol. 1 3-57
  search conditions Vol. 1 3-108
  in subqueries Vol. 1 3-114
  where Vol. 1 1-331
Exit
  unconditional, and **return** command Vol. 1 1-266 to 1-268
  waitfor command Vol. 1 1-327
exp mathematical function **Vol. 1 2-25**
Explicit null value Vol. 1 3-93
Exponent, datatype (e or E) Vol. 1 3-29
Exponential value Vol. 1 2-25

Expressions **Vol. 1 3-52** to **3-60**
definition of Vol. 1 3-52
enclosing quotes in Vol. 1 3-60
evaluation order in Vol. 1 1-314
grouping by Vol. 1 1-209
including null values Vol. 1 3-94
insert and Vol. 1 1-219
mixed mode Vol. 1 3-42
name and table name qualifying Vol. 1 3-64
numbering in mathematical functions Vol. 1 2-24
summary values for Vol. 1 1-34
types of Vol. 1 xxi, Vol. 1 3-52, Vol. 2 xix
Extended columns, Transact-SQL Vol. 1 1-211, Vol. 1 1-214
Extended data, in user-defined error messages Vol. 1 1-257
Extended datatypes, ODBC Vol. 2 2-3
Extending segments Vol. 2 1-163
Extensions
database storage Vol. 1 1-6
Transact-SQL Vol. 1 1-211
Extents Vol. 1 1-55, Vol. 1 1-82, Vol. 1 1-111

F

Failures, media
See also Recovery
automatic failover and Vol. 1 1-143
automatic unmirroring Vol. 1 3-48
disk mirroring and Vol. 1 3-47
disk remirror and Vol. 1 1-140
trunc log on chkpt database option and Vol. 2 1-120
fast option
dbcc indexalloc Vol. 1 1-112
dbcc tablealloc Vol. 1 1-111
fetch command **Vol. 1 1-192** to **1-194**
Fields, data. *See* Columns
Files
See also Tables; Transaction log
contiguous (OpenVMS) Vol. 1 1-131, Vol. 1 1-135
deleting Vol. 2 1-133
inaccessible after **sp_dropdevice** Vol. 2 1-133
localization, changing language names and Vol. 2 1-97
mirror device Vol. 1 1-134, Vol. 1 3-48
fillfactor option
alter table Vol. 1 1-12
create index Vol. 1 1-51
create table Vol. 1 1-78
Finding database objects
See also Retrieving; Search conditions
rules for Vol. 2 1-125, Vol. 2 1-173
Finding users. *See* Logins; Users
fipsflagger option, **set** Vol. 1 1-297
First column parameter. *See* Keys
First page
log device Vol. 2 1-192
text pointer Vol. 1 2-48
First-of-the-months, number of Vol. 1 2-21
fix option
dbcc indexalloc Vol. 1 1-112
dbcc tablealloc Vol. 1 1-111, Vol. 1 1-112
fix_text option, **dbcc** Vol. 1 1-113, Vol. 1 1-114
Fixed-length columns, stored order of Vol. 1 1-250
float datatype **Vol. 1 3-28**
Floating point data Vol. 1 xxi, Vol. 2 xix
str character representation of Vol. 1 2-36
Floating point datatypes. *See* Approximate numeric datatypes
floor mathematical function **Vol. 1 2-25**
flushmessage option, **set** Vol. 1 1-297
for browse option, **select** Vol. 1 1-287
for load option
alter database Vol. 1 1-7
create database Vol. 1 1-42, Vol. 1 1-44

with override Vol. 1 1-42
for read only option, **declare cursor** Vol. 1 1-119, Vol. 1 3-14
for update option, **declare cursor** Vol. 1 1-119, Vol. 1 3-14
foreign key constraint
alter table Vol. 1 1-13
create table Vol. 1 1-79
Foreign keys Vol. 1 1-82
defining Vol. 2 1-166
dropping Vol. 2 1-137
pubs2 database C-18
sp_fkeys information on Vol. 2 2-14 to 2-16
sp_helpkey and Vol. 2 1-188
syskeys table B-29
Format strings, in user-defined error messages Vol. 1 1-256
Formats, date. *See* Dates
Fragments, device space
sp_placeobject and Vol. 2 1-233
Free pages, **curunreservedpgs** system function Vol. 1 2-41
from keyword
delete Vol. 1 1-125
grant Vol. 1 1-199
joins Vol. 1 3-79
select Vol. 1 1-282
sp_tables list of objects appearing in clause Vol. 2 2-35 to 2-36
update Vol. 1 1-317
Front-end applications, browse mode and Vol. 1 3-8
Full name, modifying Vol. 2 1-220
full option
dbcc indexalloc Vol. 1 1-112
dbcc tablealloc Vol. 1 1-111
Functions
mathematical **Vol. 1 2-24** to **2-28**
string Vol. 1 2-33 to 2-39
Future space allocation. *See* **sp_placeobject** system procedure; Space allocation
futureonly option
sp_bindefault Vol. 2 1-68
sp_bindrule Vol. 2 1-75, Vol. 2 1-77
sp_unbindefault Vol. 2 1-271, Vol. 2 1-272
sp_unbindrule Vol. 2 1-276

G

Gaps in IDENTITY column values Vol. 1 3-75 to 3-77
German language print message example Vol. 1 1-253
getdate date function **Vol. 1 2-19**
Getting messages. *See* **sp_getmessage** system procedure
Global audit options, *sysauditoptions* system table B-6
Global variables **Vol. 1 3-143** to **3-149**
See also individual variable names
sp_monitor report on Vol. 2 1-227
go command terminator Vol. 1 3-6
goto keyword **Vol. 1 1-195**
Grammatical structure, numbered place holders and Vol. 1 1-253
Grand totals
compute Vol. 1 1-35
order by Vol. 1 1-249
grant command **Vol. 1 1-196** to **1-206**
auditing use of Vol. 2 1-52
and roles Vol. 1 1-198, Vol. 1 1-204
sysprotects table B-45
grant option
sp_helprotect Vol. 2 1-195
sp_role Vol. 2 1-256
grant option for option, **revoke** Vol. 1 1-271
Granting roles with **sp_role** Vol. 2 1-256 to 1-257
Greater than. *See* Comparison operators
Greek characters Vol. 1 3-65
group by clause **Vol. 1 1-207** to **1-216**
aggregate functions and Vol. 1 1-207, Vol. 1 1-209, Vol. 1 2-3, Vol. 1 2-6
cursors and Vol. 1 3-20
having clause and Vol. 1 1-207 to 1-216
having clause and, in standard SQL Vol. 1 1-210

having clause and, in Transact-SQL Vol. 1 1-211
having clause and, sort orders Vol. 1 1-215
null values and Vol. 1 3-93
select Vol. 1 1-284 to 1-285
views and Vol. 1 1-108
without **having** clause Vol. 1 1-213
Grouping
See also User-defined transactions
multiple trigger actions Vol. 1 1-94
procedures Vol. 1 3-125 to 3-141
procedures of the same name Vol. 1 1-58, Vol. 1 1-152, Vol. 1 1-187
table rows Vol. 1 1-210
Groups
See also "public" group
changing Vol. 2 1-81 to 1-82
dropping Vol. 2 1-135 to 1-136
grant and Vol. 1 1-205
information on Vol. 2 1-182
revoke and Vol. 1 1-273
sp_addgroup Vol. 2 1-13 to 1-14
sp_adduser procedure Vol. 2 1-49
sysusers table entries for B-59
table rows Vol. 1 1-207
Guest accounts Vol. 1 1-205, B-59, Vol. 2 1-158
Guest users
in *sybsystemprocs* database Vol. 2 1-6

H

Halloween problem Vol. 1 1-123, Vol. 1 3-21
having clause **Vol. 1 1-207** to **1-216**
aggregate functions and Vol. 1 1-208, Vol. 1 1-209, Vol. 1 2-2, Vol. 1 2-4
difference from **where** clause Vol. 1 3-105
group by and Vol. 1 1-207 to 1-216
group by extensions in Transact-SQL and Vol. 1 1-211
negates **all** Vol. 1 1-208
select Vol. 1 1-285
subqueries in outer query Vol. 1 3-115
Headings, column Vol. 1 1-208, Vol. 1 1-209
in views Vol. 1 1-103
Help
sp_syntax display Vol. 2 1-265
Technical Support Vol. 1 xxii, Vol. 2 xx
Help reports
See also Information (Server); System procedures
database object Vol. 2 1-171
datatypes Vol. 1 3-41
system procedures Vol. 2 1-171 to 1-209
Hexadecimal numbers, “0x” prefix for Vol. 1 1-46
hextoint function Vol. 1 2-9, Vol. 1 2-17
hh. *See* **hour** date part
Hierarchy
See also Precedence
datatype Vol. 1 3-43, B-55
operators Vol. 1 3-53
user-defined datatypes Vol. 2 1-42
Hierarchy of permissions. *See* Permissions
Historic dates, pre-1753 Vol. 1 2-22
holdlock keyword
cursors and Vol. 1 3-23
readtext Vol. 1 1-261
select Vol. 1 1-283, Vol. 2 1-213
Host computer name Vol. 1 2-42
Host process ID, client process Vol. 1 2-42
host_id system function Vol. 1 2-42
host_name system function Vol. 1 2-42
hour date part Vol. 1 2-20
Hour values date style Vol. 1 2-10
Hyphens as comments Vol. 1 3-11

I

I/O devices, disk mirroring to Vol. 1 1-134, Vol. 1 3-48

Identifiers **Vol. 1 3-61** to **3-66**
changing names of Vol. 2 1-90
quoting Vol. 2 1-89
renaming Vol. 1 3-65
reserved words and Vol. 2 1-85 to 1-99
select Vol. 1 1-288
set quoted_identifiers on Vol. 2 1-89, Vol. 2 1-97 to 1-98
sp_checkreswords and Vol. 2 1-89
system functions and Vol. 1 2-44
Identities
setuser command Vol. 1 1-306
users Vol. 1 3-85 to 3-87
IDENTITY columns **Vol. 1 3-67** to **3-78**
alter table and Vol. 1 3-67
bulk copy and Vol. 1 3-73
create table and Vol. 1 1-91, Vol. 1 3-67
creating Vol. 1 3-67
datatype of Vol. 1 3-67
gaps in values Vol. 1 3-75 to 3-77
insert and Vol. 1 1-223 to 1-224
inserting explicit values Vol. 1 3-72
inserting tables with Vol. 1 1-219, Vol. 1 3-71
inserting with **alter table** Vol. 1 1-17
maximum value of Vol. 1 3-73
null values and Vol. 1 3-88
retrieving last value Vol. 1 3-72
select and Vol. 1 1-289 to 1-291
selecting Vol. 1 3-68
set identity_insert on Vol. 1 3-72
sp_addtype and Vol. 2 1-40
system-generated values Vol. 1 3-71
unique values for Vol. 1 3-72
update and Vol. 1 1-320
updates not allowed Vol. 1 1-320
values Vol. 1 1-223
views and Vol. 1 1-107, Vol. 1 3-73 to 3-74
identity configuration variable Vol. 1 1-223, Vol. 1 3-76
@@identity global variable Vol. 1 1-224, Vol. 1 3-72 to 3-73, Vol. 1 3-146
identity keyword **Vol. 1 3-67**
alter table Vol. 1 1-11
Identity of user. *See* Aliases; Logins; Users
IDENTITY property
sp_help and Vol. 1 3-70
user-defined datatype specification and Vol. 1 3-40, Vol. 1 3-74
identity_insert option, **set** Vol. 1 1-297
@@idle global variable Vol. 1 3-146
sp_monitor and Vol. 2 1-228
IDs, server role (*sysroles* table) B-50
IDs, user
See also Logins
database (**db_id**) Vol. 1 2-41
server user Vol. 1 2-43
stored procedure (**procid**) Vol. 1 1-298
user name Vol. 1 2-43
user_id function for Vol. 1 2-43
if update clause, **create trigger** Vol. 1 1-94, Vol. 1 1-95, Vol. 1 1-100
if...else conditions **Vol. 1 1-217** to **1-218**
continue and Vol. 1 1-39
local variables and Vol. 1 1-118
ignore_dup_key option, **create index** Vol. 1 1-52
ignore_dup_row option, **create index** Vol. 1 1-52
image datatype Vol. 1 3-32, **Vol. 1 3-120** to **3-124**
"0x" prefix for Vol. 1 3-123
functions Vol. 1 2-48 to 2-50
initializing Vol. 1 3-120
initializing with null values Vol. 1 3-93
null values in Vol. 1 3-93, Vol. 1 3-121
pointer values in **readtext** Vol. 1 1-261
storage on separate devices Vol. 1 1-261
sysindexes table and B-26
writetext to Vol. 1 1-338
Immediate shutdown Vol. 1 1-308
Impersonating a user. *See* **setuser** command
Implicit conversion (of datatypes) Vol. 1 3-44, Vol. 1 3-59

in keyword
alter table and Vol. 1 1-14
check constraint using Vol. 1 1-91
in expressions Vol. 1 3-57
search conditions Vol. 1 3-108
subqueries Vol. 1 3-113
where Vol. 1 1-332
Inactive transaction log space Vol. 1 1-175
Included groups, **group by** query Vol. 1 1-211
Incompatibility of data. *See* Character set conversion; Conversion
Index pages
allocation of Vol. 1 2-43
fillfactor values for Vol. 1 1-12, Vol. 1 1-51, Vol. 1 1-78
leaf level Vol. 1 1-12, Vol. 1 1-50, Vol. 1 1-51, Vol. 1 1-78
system functions Vol. 1 2-41, Vol. 1 2-43, Vol. 1 2-46
total of table and Vol. 1 2-43
index_col system function Vol. 1 2-42
indexalloc option, **dbcc** Vol. 1 1-112
Indexes
See also Clustered indexes; Database objects; Non-clustered indexes
changing names of Vol. 2 1-92
checking name with **sp_checkreswords** Vol. 2 1-88
checking with **sp_checknames** Vol. 2 1-83
create index Vol. 1 1-50 to 1-57
cursors using Vol. 1 3-19 to 3-21
dbcc indexalloc and Vol. 1 1-112
dropping Vol. 1 1-150 to 1-151
estimating space and time requirements Vol. 2 1-159
on IDENTITY columns Vol. 1 3-72
information on Vol. 2 1-184
on joined columns Vol. 1 1-55
listing Vol. 1 1-150
naming Vol. 1 1-51
non-clustered Vol. 1 1-51
page allocation check Vol. 1 1-112
reindex integrity check Vol. 1 1-112 to 1-113
renaming Vol. 2 1-248 to 1-250
sp_placeobject space allocation for Vol. 2 1-233 to 1-234
sp_statistics information on Vol. 2 2-28 to 2-29
space used by Vol. 2 1-263
splitting Vol. 2 1-233
subqueries using Vol. 1 3-114
suspect Vol. 2 1-210 to 1-211
sysindexes table Vol. 1 3-121
system tables entries for B-26
truncate table and Vol. 1 1-311
types of Vol. 1 1-50 to 1-51, Vol. 1 1-53
update statistics on Vol. 1 1-324
views and Vol. 1 1-55
Indirection between index structure and data Vol. 1 1-51
Infected processes
removal with **kill** Vol. 2 1-287
waitfor errorexit and Vol. 1 1-328
Information (Server)
alternate languages Vol. 2 1-190
configuration variables B-16, B-19
current locks Vol. 2 1-212
database devices Vol. 2 1-180
database objects Vol. 2 1-171
Database Owners Vol. 2 1-208 to 1-209
databases B-20 to B-21, Vol. 2 1-178
datatypes Vol. 1 3-41, Vol. 2 1-171
display procedures Vol. 1 1-60
dump devices Vol. 2 1-180
first page of log Vol. 2 1-192
groups Vol. 2 1-182, Vol. 2 1-208 to 1-209
indexes Vol. 2 1-184
join columns Vol. 2 1-186
keys Vol. 2 1-188
languages Vol. 2 1-190
log device Vol. 2 1-192
logins Vol. 2 1-286 to 1-287
monitor statistics Vol. 2 1-227

remote server logins Vol. 2 1-193
remote servers Vol. 2 1-200
segments Vol. 2 1-198
server logins Vol. 2 1-286 to 1-287
space usage Vol. 1 1-56, Vol. 2 1-263
suspect indexes Vol. 2 1-210 to 1-211
text Vol. 1 1-65, Vol. 2 1-204
thresholds Vol. 2 1-206
user permissions Vol. 2 1-195
users Vol. 2 1-129
users, database Vol. 2 1-208 to 1-209
Information messages (Server). *See* Error messages; Severity levels
Initializing
disk Vol. 1 1-130 to 1-133
disk reinit and Vol. 1 1-132, Vol. 1 1-138 to 1-139
text or *image* columns Vol. 1 3-122
In-memory map Vol. 1 1-8
Inner queries. *See* Nesting; Subqueries
Inner table of an outer join Vol. 1 3-82
Input packets, number of Vol. 1 3-147, Vol. 2 1-228
insert command **Vol. 1 1-219** to **1-226**
auditing use of Vol. 2 1-58
bound rules and Vol. 2 1-76
create default and Vol. 1 1-46
create procedure and Vol. 1 1-63
IDENTITY columns and Vol. 1 1-223 to 1-224, Vol. 1 3-71
null / not null columns and Vol. 1 1-107, Vol. 1 3-91, Vol. 1 3-94
triggers and Vol. 1 1-97, Vol. 1 1-100
update and Vol. 1 1-220
views and Vol. 1 1-107, Vol. 1 1-224 to 1-225
inserted table
triggers and Vol. 1 1-96, Vol. 1 1-98
Inserting
See also Padding, data
leading zero automatic Vol. 1 3-32
spaces in text strings Vol. 1 2-35
int datatype **Vol. 1 3-27**
aggregate functions and Vol. 1 2-7
Integer data Vol. 1 3-27
in SQL Vol. 1 xxi, Vol. 2 xix
Integer datatypes, converting to Vol. 1 2-16
Integer remainder. *See* Modulo operator (%)
Integrity constraints Vol. 1 1-84
types Vol. 1 1-85
Integrity of data
transactions and Vol. 1 3-139
Integrity. *See* **dbcc** (Database Consistency Checker); Referential integrity
Intent table locks Vol. 2 1-213
Interfaces file
changing server names in Vol. 2 1-96
sp_addserver and Vol. 2 1-31
Internal datatypes of null columns Vol. 1 1-83, Vol. 1 3-44
See also Datatypes
Internal structures, pages used for Vol. 1 2-41, Vol. 1 2-43
Interval, automatic checkpoint Vol. 1 1-25
into clause, **select**. *See* **select into** command
into keyword
declare cursor Vol. 1 3-15
fetch Vol. 1 1-192
insert Vol. 1 1-219
union Vol. 1 1-313
inttohex function Vol. 1 2-9, Vol. 1 2-17
@@io_busy global variable Vol. 1 3-146
sp_monitor and Vol. 2 1-228
is not null keyword in expressions Vol. 1 3-58
is null keyword Vol. 1 3-92
in expressions Vol. 1 3-58
where Vol. 1 1-332
isnull system function Vol. 1 2-42, **Vol. 1 3-92**
insert and Vol. 1 1-222, Vol. 1 3-94
print and Vol. 1 1-255
select and Vol. 1 1-288
ISO character sets Vol. 1 3-65

@@isolation global variable Vol. 1 3-133, Vol. 1 3-146
Isolation levels
  cursor locking Vol. 1 3-24
  transactions Vol. 1 3-132 to 3-133
isql utility command
  approximate numeric datatypes and Vol. 1 3-28
  defaults and Vol. 1 1-46
  go command terminator Vol. 1 3-6

J

Japanese character sets
  object identifiers and Vol. 1 3-65
  print message example Vol. 1 1-253
Joins **Vol. 1 3-79** to **3-84**
  connecting Vol. 1 3-80
  count or **count(*)** with Vol. 1 2-7
  equijoins Vol. 1 3-81
  information on Vol. 2 1-186
  not-equal Vol. 1 3-81
  null column values and Vol. 1 3-90
  null values and Vol. 1 3-83
  operators for Vol. 1 3-80
  outer Vol. 1 3-82
  restrictions Vol. 1 3-80
  self-joins Vol. 1 3-82
  sp_commonkey Vol. 2 1-102
  subqueries compared to Vol. 1 3-82
  table groups and Vol. 1 1-214
  theta Vol. 1 3-80

K

Key values, distribution of Vol. 1 1-324
Keys, table Vol. 1 1-82
  See also Common keys; Indexes
  dropping Vol. 2 1-137
  information on Vol. 2 1-188
  pubs2 database primary and foreign C-18
  syskeys table B-29 to B-30, Vol. 2 1-102, Vol. 2 1-166, Vol. 2 1-235
  system B-29
Keywords **A-1** to **A-6**
  APT-SQL A-3
  control-of-flow Vol. 1 3-12
  as identifiers Vol. 2 1-85
  Transact-SQL Vol. 1 3-61, A-1 to A-2
kill command **Vol. 1 1-227** to **1-229**
  sp_who and Vol. 2 1-286, Vol. 2 1-287

L

Labels
  dump volumes Vol. 1 1-170, Vol. 1 1-236, Vol. 1 1-243
  goto label Vol. 1 1-195
  tape Vol. 1 1-170
@@langid global variable Vol. 1 1-256, Vol. 1 3-146
Language
  setting system messages Vol. 1 1-297
  sp_getmessage Vol. 2 1-169
Language cursors Vol. 1 3-18
Language defaults Vol. 2 1-20, Vol. 2 1-21
  adding Vol. 2 1-15 to 1-19
  weekday order and Vol. 1 2-22
@@language global variable Vol. 1 3-146
language option, **set** Vol. 1 1-297
Languages, alternate
  alias for Vol. 2 1-261
  changing names of Vol. 2 1-94, Vol. 2 1-97
  checking with **sp_checkreswords** Vol. 2 1-89
  date formats in Vol. 2 1-15
  dropping Vol. 2 1-140 to 1-141
  dropping messages in Vol. 2 1-144
  information on Vol. 2 1-190
  month and day names in Vol. 2 1-15
  official name Vol. 2 1-261
  structure and translation Vol. 1 1-253
  syslanguages table B-31, Vol. 2 1-190
  system tables entries for B-31

Last-chance thresholds Vol. 1 2-42, Vol. 2 1-35, Vol. 2 1-223, Vol. 2 1-225
lct_admin system function **Vol. 1 2-42**
Leading blanks, removal with **ltrim** function Vol. 1 2-34
Leading zeros, automatic insertion of Vol. 1 3-32
Leaf levels of indexes
clustered index Vol. 1 1-12, Vol. 1 1-50, Vol. 1 1-51, Vol. 1 1-78
Leaving a procedure. *See* **return** command
Length
See also Size
defining column Vol. 1 2-41
of expressions in bytes Vol. 1 2-41
of columns Vol. 1 2-41
Less than. *See* Comparison operators
Levels
nesting Vol. 1 1-66, Vol. 1 3-127
nesting triggers Vol. 1 1-100
@@nestlevel Vol. 1 1-190, Vol. 1 3-147
permission assignment Vol. 1 1-199
@@trancount global variable Vol. 1 3-127, Vol. 1 3-148
transaction isolation Vol. 1 3-132 to 3-133
like keyword
alter table and Vol. 1 1-14
check constraint using Vol. 1 1-91
in expressions Vol. 1 3-57
search conditions and Vol. 1 3-106
searching for dates with Vol. 1 3-38
where Vol. 1 1-332
wildcard characters used with Vol. 1 3-58
Linkage, page. *See* Pages (data)
Linking users. *See* Aliases
List
catalog stored procedures Vol. 2 2-1
commands requiring roles Vol. 1 3-101
configuration variables Vol. 2 1-106 to 1-109
database auditing options Vol. 2 1-52
error return values Vol. 1 1-267
global variables Vol. 1 3-145 to 3-149
mathematical functions Vol. 1 2-25 to 2-28
reserved return status values Vol. 1 1-267
sort order choices and effects Vol. 1 1-250
system procedures Vol. 2 1-1 to 1-6
system tables B-1 to B-3
Listing
datatypes with types Vol. 1 3-43
existing defaults Vol. 1 1-148
user group members Vol. 1 1-205
Literal character specification
like match string Vol. 1 3-153
quotes (" ") Vol. 1 2-40, Vol. 1 3-60
Literal values
datatypes of Vol. 1 3-42
in expressions Vol. 1 2-40
null Vol. 1 3-94
load database command **Vol. 1 1-230** to **1-237**
load striping Vol. 1 1-231
messages Vol. 1 1-236, Vol. 1 1-244
load transaction command **Vol. 1 1-238** to **1-245**
Load, database Vol. 1 1-230 to 1-237
across networks Vol. 1 1-235, Vol. 1 1-236
Backup Server and Vol. 1 1-236
commands used for Vol. 1 1-233, Vol. 1 1-241
cross-platform not supported Vol. 1 1-234, Vol. 1 1-242
disk mirroring and Vol. 1 1-236
dismounting tapes after Vol. 1 1-231
file name Vol. 1 1-231
file name, listing Vol. 1 1-232
header, listing Vol. 1 1-232
load striping Vol. 1 1-231
message destination Vol. 1 1-232
new database Vol. 1 1-44
remote Vol. 1 1-235

restricting use Vol. 1 1-235, Vol. 1 1-242
restrictions Vol. 1 1-234
rewinding tapes after Vol. 1 1-231
size required Vol. 1 1-234
updates prohibited during Vol. 1 1-234
volume name Vol. 1 1-231
Load, transaction log Vol. 1 1-238 to 1-245
commands used for Vol. 1 1-233, Vol. 1 1-241
cross-platform not supported Vol. 1 1-242
disk mirroring and Vol. 1 1-244
dismounting tape after Vol. 1 1-239
dump devices Vol. 1 1-239
file name Vol. 1 1-239
file name, listing Vol. 1 1-240
header, listing Vol. 1 1-240
load striping Vol. 1 1-239
message destination Vol. 1 1-240
rewinding tape after Vol. 1 1-239
Timestamps Vol. 1 1-242
volume name Vol. 1 1-239
Local alias, language Vol. 2 1-261
local option, **sp_addserver** Vol. 2 1-31
Local servers Vol. 2 1-31
See also Remote servers; Servers
Local variables **Vol. 1 3-143** to **3-149**
declare (name and datatype) Vol. 1 1-117, Vol. 1 3-41
raiserror and Vol. 1 1-257
in screen messages Vol. 1 1-254
in user-defined error messages Vol. 1 1-257
Localization files
changing language names and Vol. 2 1-97
Location of new database Vol. 1 1-41
lock | unlock option, **sp_locklogin** Vol. 2 1-215
Locking
control over Vol. 2 1-212 to 1-214
login account status Vol. 2 1-129
login accounts Vol. 2 1-215
text for reads Vol. 1 1-261
Locking and cursors Vol. 1 3-22 to 3-24
levels Vol. 1 3-24
Locks
exclusive Vol. 2 1-213
intent table Vol. 2 1-213
shared Vol. 2 1-213
sp_lock system procedure Vol. 2 1-212 to 1-214
syslocks table B-33
system tables entries for B-33
types of Vol. 2 1-213
Log device
See also Transaction logs
information Vol. 2 1-192
purging a Vol. 1 1-169
space allocation Vol. 1 1-44, Vol. 1 1-114, Vol. 1 1-131
log mathematical function **Vol. 1 2-26**
log on option
alter database Vol. 1 1-6
create database Vol. 1 1-42
create database, and sp_logdevice Vol. 2 1-217
Log segment
dbcc checktable report on Vol. 1 1-110
not on its own device Vol. 1 1-111
sp_threshold report on Vol. 2 1-206
log10 mathematical function **Vol. 1 2-26**
Logarithm, base 10 Vol. 1 2-26
Logging *text* or *image* data Vol. 1 1-338
Logical (conceptual) tables Vol. 1 1-96, Vol. 1 1-98
Logical consistency. *See* **dbcc** (Database Consistency Checker)
Logical device name Vol. 2 1-46, Vol. 2 1-127
disk mirroring Vol. 1 1-134
disk remirroring Vol. 1 1-140
disk unmirroring Vol. 1 1-143
for *syslogs* table Vol. 2 1-217
new database Vol. 1 1-41

Logical expressions Vol. 1 xxi, Vol. 2 xix
if...else Vol. 1 1-217
syntax Vol. 1 1-23, Vol. 1 3-52
truth tables for Vol. 1 3-58 to 3-59
Logical reads, **statistics io** Vol. 1 1-299
Login management **Vol. 1 3-85** to **3-87**
Logins
See also Remote logins; Users
accounting statistics Vol. 2 1-100, Vol. 2 1-255
adding to servers Vol. 2 1-20 to 1-22
alias Vol. 2 1-9, Vol. 2 1-131
auditing Vol. 1 3-3, Vol. 2 1-62
changing current database owner Vol. 2 1-79
char_convert setting for Vol. 1 1-296
disabling Vol. 1 1-309
dropping remote Vol. 2 1-146 to 1-147
dropping with **sp_droplogin** system procedure Vol. 2 1-142
information on Vol. 2 1-129, Vol. 2 1-193
locking and unlocking accounts Vol. 2 1-215
management Vol. 1 3-85 to 3-87
modifying accounts Vol. 2 1-220 to 1-221
number of Vol. 2 1-228
options for remote Vol. 2 1-245
password change Vol. 2 1-230 to 1-232
sp_dropserver droplogins procedure Vol. 2 1-151
syslogins table B-36 to B-37
sysremotelogins table B-49, Vol. 2 1-25 to 1-27, Vol. 2 1-146, Vol. 2 1-151, Vol. 2 1-193
Logs. *See* Segments; Transaction logs
logsegment log storage
dropping Vol. 2 1-149
log10 mathematical function **Vol. 1 2-26**
Loops
goto label Vol. 1 1-195
syslogs changes and infinite B-38
trigger chain infinite Vol. 1 1-100
while Vol. 1 1-23, Vol. 1 1-336
while, continue and Vol. 1 1-39
while, local variables and Vol. 1 3-143
Lower and higher datatypes. *See* Precedence
lower string function Vol. 1 2-34
Lowercase letters, sort order and Vol. 1 1-250
See also Case sensitivity
ltrim string function Vol. 1 2-34

M

Machine ticks Vol. 2 1-228
Macintosh character set Vol. 1 3-65
Mapping
See also Aliases
databases Vol. 2 1-122
remote server users Vol. 2 1-25
system and default segments Vol. 1 1-9
sysusages table B-57
Markers, user-defined. *See* Place holders; Savepoints
master database
alter database and Vol. 1 1-7
backing up Vol. 1 1-180
backup after **disk init** Vol. 1 1-132
backup after **disk mirror** Vol. 1 1-135
backup after **disk remirror** Vol. 1 1-140
backup after **disk unmirror** Vol. 1 1-144
checking with **sp_checkreswords** Vol. 2 1-88
disk refit and Vol. 1 1-137
disk reinit and Vol. 1 1-138
drop index and Vol. 1 1-150
dropping databases and Vol. 1 1-146
dropping user-created system procedures from Vol. 1 1-152
loading a backup Vol. 1 1-236, Vol. 1 1-244
sp_dboption and Vol. 2 1-117
system procedure tables Vol. 2 1-8
system tables B-2

thresholds and Vol. 2 1-36, Vol. 2 1-224
transaction log purging Vol. 1 1-168, Vol. 1 1-181
master database recovery. *See* Recovery of *master* database
Master device Vol. 1 1-8
Matching
See also Comparison; Pattern matching
name and table name Vol. 1 3-64
row (*= or =*), outer join Vol. 1 3-82
values in joins Vol. 1 3-79 to 3-84
Mathematical functions **Vol. 1 2-24** to **2-28**
rand Vol. 1 2-28
syntax Vol. 1 2-24
max aggregate function **Vol. 1 2-3**
as row aggregate Vol. 1 2-29
@@max_connections global variable Vol. 1 3-146
@@maxcharlen global variable Vol. 1 3-146
Memory
See also Space
mapping Vol. 2 1-122
releasing with **deallocate cursor** Vol. 1 1-116
Message number Vol. 2 1-23, Vol. 2 1-144, Vol. 2 1-169
Message **output** parameter, **sp_getmessage** Vol. 2 1-169
Messages
adding user-defined Vol. 2 1-23 to 1-24
dropping system with **sp_droplanguage** Vol. 2 1-140
dropping user-defined Vol. 2 1-144 to 1-145
language setting for Vol. 1 1-297, Vol. 2 1-144, Vol. 2 1-169
load transaction Vol. 1 1-242
mathematical functions and Vol. 1 2-28
printing user-defined Vol. 1 1-253 to 1-255
revoke Vol. 1 1-274
screen Vol. 1 1-253 to 1-255
sp_getmessage procedure Vol. 2 1-169 to 1-170
sp_volchanged list Vol. 2 1-282 to 1-285
sysmessages table B-39
sysusermessages table **B-58**, Vol. 2 1-23 to 1-24
transactions and Vol. 1 3-128
trigger Vol. 1 1-97
unbinding with **sp_unbindmsg** Vol. 2 1-274 to 1-275
user-defined B-58
Messages, system procedure. *See* System procedures; *individual procedure names*
mi. *See* **minute** date part
Midnights, number of Vol. 1 2-21
Migration
of system log to another device Vol. 1 1-132
of tables to clustered indexes Vol. 1 1-55, Vol. 1 1-84
millisecond date part Vol. 1 2-20
Millisecond values
datediff results in Vol. 1 2-22
input and storage of Vol. 1 3-36
min aggregate function **Vol. 1 2-3**
as row aggregate Vol. 1 2-29
Minus sign (-) subtraction operator Vol. 1 3-53
minute date part Vol. 1 2-20
Minute values date style Vol. 1 2-10
mirror keyword, **disk mirror** Vol. 1 1-134, Vol. 1 3-48
mirrorexit keyword
waitfor Vol. 1 1-327
Mirroring. *See* Disk mirroring
Mistakes, user. *See* Errors
Mixed datatypes, arithmetic operations on Vol. 1 3-42, Vol. 1 3-53
mm. *See* **month** date part

mode option, **disk unmirror** Vol. 1 1-143, Vol. 1 3-49
model database
copying the Vol. 1 1-42
sp_dboption and Vol. 2 1-117
user-defined datatypes in Vol. 1 3-41
Modules, display syntax of Vol. 2 1-265
Modulo operator (**%**) Vol. 1 3-53
use restrictions Vol. 1 3-53
Monetary datatypes **Vol. 1 3-34**
arithmetic operations and Vol. 1 3-34
money datatype **Vol. 1 3-34**
Money symbols Vol. 1 3-61
Monitor SQL statistics
global variables and **sp_monitor** Vol. 1 3-145
sp_monitor Vol. 2 1-227
Monitoring
space remaining Vol. 2 1-34, Vol. 2 1-35, Vol. 2 1-223
month date part Vol. 1 2-20
Month values
alternate language B-31, Vol. 2 1-15
date part abbreviation and Vol. 1 2-20
date style Vol. 1 2-10
entering in alphabetic format Vol. 1 3-36
short (abbreviated) B-31
syslanguages table B-31
ms. *See* **millisecond** date part
Multibyte character sets
converting Vol. 1 2-14
fix_text upgrade for Vol. 1 1-113, Vol. 1 1-114
identifier names Vol. 1 3-65
nchar datatype for Vol. 1 3-29
readtext and Vol. 1 1-262
readtext using characters for Vol. 1 1-263
sort order Vol. 2 1-203
sp_helpsort output Vol. 2 1-203
wildcard characters and Vol. 1 3-153
writetext and Vol. 1 1-339
Multiple trigger actions Vol. 1 1-94
Multiple-line comments Vol. 1 3-10
Multiplication (*****) operator Vol. 1 3-53
Multi-table views Vol. 1 1-107, Vol. 1 1-322, Vol. 1 3-83
See also Views
delete and Vol. 1 1-107, Vol. 1 1-126, Vol. 1 3-83
insert and Vol. 1 3-83

N

"N / A", using "NULL" or Vol. 1 3-93
Name of device
disk mirroring and Vol. 1 1-134
disk remirroring and Vol. 1 1-140
disk unmirroring and Vol. 1 1-143
dump device Vol. 1 1-163, Vol. 1 1-176
physical, **disk reinit** and Vol. 1 1-138
name option
disk init Vol. 1 1-130
disk reinit Vol. 1 1-138
Names
See also Identifiers
alias Vol. 1 1-283, Vol. 2 1-9, Vol. 2 1-131, Vol. 2 1-157
assigning different, compared to aliases Vol. 2 1-49
changing database object Vol. 2 1-248 to 1-250
changing identifier Vol. 2 1-90
character set B-11
checking with **sp_checknames** Vol. 2 1-83
checking with **sp_checkreswords** Vol. 2 1-85
checking with **valid_name** Vol. 1 3-65
column, in views Vol. 1 1-103
configuration variables Vol. 2 1-106 to 1-109
of date parts Vol. 1 2-20
db_name function Vol. 1 2-42
finding non-duplicate in joins Vol. 1 3-82
finding similar-sounding Vol. 1 2-38
host computer Vol. 1 2-42

index_col and index Vol. 1 2-42
object_name function Vol. 1 2-42
omitted elements of (..) Vol. 1 3-63
parameter, in **create procedure** Vol. 1 1-58
qualifying database objects Vol. 1 3-63, Vol. 1 3-65
remote user Vol. 2 1-146
segment Vol. 1 1-13, Vol. 1 1-54, Vol. 1 1-55, Vol. 1 1-79, Vol. 1 1-80, Vol. 2 1-29
server Vol. 2 1-31
server attribute Vol. 2 2-19
setuser Vol. 1 1-306
sort order B-11
sorting groups of Vol. 1 1-215
suser_name function Vol. 1 2-43
of transactions Vol. 1 3-131
user system function Vol. 1 2-43
user_name function Vol. 1 2-44
view Vol. 1 1-160
weekday numbers and Vol. 1 2-22
Names in calendar. *See* Date parts
Naming
columns in views Vol. 1 1-103
conventions Vol. 1 3-61 to 3-66
cursors Vol. 1 1-120
database device Vol. 1 1-130
database objects Vol. 1 3-61 to 3-66
file Vol. 1 1-130
identifiers Vol. 1 3-61 to 3-66
indexes Vol. 1 1-51
stored procedures Vol. 1 1-63
tables Vol. 1 1-75
temporary tables Vol. 1 1-75, Vol. 1 3-116
transactions Vol. 1 3-125
triggers Vol. 1 1-94
user-defined datatypes Vol. 1 3-40, Vol. 2 1-42
views Vol. 1 1-103
National Character. *See nchar* datatype
Natural joins Vol. 1 3-81
Natural logarithm Vol. 1 2-26
nchar datatype **Vol. 1 3-29** to **3-30**
@@ncharsize global variable Vol. 1 3-147
sp_addtype and Vol. 2 1-42
Negative (-) sign for monetary values Vol. 1 3-34
Nested **select** statements. *See* **select** command; Subqueries
nested triggers configuration variable Vol. 1 1-100, Vol. 1 1-101
Nesting
See also Joins
aggregate functions Vol. 1 2-5
begin transaction/commit statements Vol. 1 3-127
begin...end blocks Vol. 1 1-19
comments Vol. 1 3-10
cursors Vol. 2 1-111
if tests Vol. 1 1-218
levels Vol. 1 1-66
levels of triggers Vol. 1 1-100
stored procedures Vol. 1 1-63, Vol. 1 1-190
string functions Vol. 1 2-33, Vol. 1 2-37
subqueries Vol. 1 3-110 to 3-115
transaction statements Vol. 1 3-127
triggers Vol. 1 1-100
warning on transactions Vol. 1 3-131
while loops Vol. 1 1-337
while loops, **break** and Vol. 1 1-24
@@nestlevel global variable Vol. 1 1-66, Vol. 1 1-101, Vol. 1 1-190, Vol. 1 3-147
net password encryption option
sp_serveroption Vol. 2 1-258
New database, creating Vol. 1 1-41
%*nn*! (place holder format) Vol. 1 1-253
no chkpt on recovery database option
setting with **sp_dboption** Vol. 2 1-119
no free space acctg database option
setting with **sp_dboption** Vol. 2 1-118
nocount option, **set** Vol. 1 1-297
nodismount option
dump database Vol. 1 1-164
dump transaction Vol. 1 1-177
noexec option, **set** Vol. 1 1-297

noholdlock keyword, **select** Vol. 1 1-284
nonclustered constraint
alter table Vol. 1 1-12
create table Vol. 1 1-78
Non-clustered indexes Vol. 1 1-51
"none", using "NULL" or Vol. 1 3-93
Non-logged operations Vol. 1 1-338
Non-repeatable reads Vol. 1 3-133
noserial option, **disk mirror** Vol. 1 1-134, Vol. 1 3-48
Not equal joins (**!=** or **<>**) Vol. 1 3-81
not in keyword
subqueries using Vol. 1 3-114
not keyword
in expressions Vol. 1 3-57
in joins Vol. 1 3-81
search conditions Vol. 1 3-106
where Vol. 1 1-333
not like keyword Vol. 1 3-151
not null keyword Vol. 1 3-88
create table Vol. 1 1-77
in expressions Vol. 1 3-58
Not null values
defining Vol. 1 1-48, Vol. 1 3-92
dropping defaults for Vol. 1 1-148
insert and Vol. 1 1-222
search conditions Vol. 1 3-107
select statements and Vol. 1 1-288
sp_addtype and Vol. 2 1-41
spaces in Vol. 1 3-31
for user-defined data Vol. 2 1-41
views and Vol. 1 1-107
nowait option, **shutdown** Vol. 1 1-308
null keyword Vol. 1 3-88
alter table Vol. 1 1-11
create table Vol. 1 1-77
in expressions Vol. 1 3-58
Null string in character columns Vol. 1 2-38, Vol. 1 3-93
Null types, user-defined datatype specification and Vol. 1 3-40
Null values **Vol. 1 3-88** to **3-95**
column datatype conversion for Vol. 1 3-31
column defaults and Vol. 1 1-48, Vol. 1 1-70
comparing Vol. 1 1-294
create procedure and Vol. 1 3-91
default parameters as Vol. 1 3-90
defining Vol. 1 1-48, Vol. 1 1-82, Vol. 1 3-92
dropping defaults for Vol. 1 1-148
in expressions Vol. 1 3-58
group by and Vol. 1 1-209
inserting into *text* and *image* columns Vol. 1 1-221
inserting substitute values for Vol. 1 1-222, Vol. 1 3-94
joins and Vol. 1 3-83
new column Vol. 1 1-11, Vol. 1 1-48
new rules and column definition Vol. 1 1-70, Vol. 1 3-94
not allowed in IDENTITY columns Vol. 1 3-67
null defaults and Vol. 1 1-48, Vol. 1 1-70, Vol. 1 3-92
in search conditions Vol. 1 3-107
select statements and Vol. 1 1-288
sort order of Vol. 1 1-249, Vol. 1 3-94
sp_addtype and Vol. 2 1-40
stored procedures cannot return Vol. 1 1-268
text and *image* columns Vol. 1 3-120
triggers and Vol. 1 1-100
for user-defined datatypes Vol. 2 1-40
Number (quantity of)
See also Range; Size
active dumps or loads Vol. 1 1-169, Vol. 1 1-183, Vol. 1 1-235, Vol. 1 1-243
arguments and place holders Vol. 1 1-254
arguments, in a **where** clause Vol. 1 1-335
bytes in returned text Vol. 1 1-262
bytes per row Vol. 1 1-15, Vol. 1 1-82
clustered indexes Vol. 1 1-50

databases Server can manage Vol. 1 1-42
databases within transactions Vol. 1 3-128
device fragments Vol. 1 1-8, Vol. 1 1-42
different triggers Vol. 1 1-97
first-of-the-months Vol. 1 2-21
groups per user Vol. 2 1-81
having clause search arguments Vol. 1 1-208
logical reads (**statistics io**) Vol. 1 1-299
messages per constraint Vol. 2 1-72
midnights Vol. 1 2-21
named segments Vol. 1 1-43, Vol. 2 1-29
nesting levels Vol. 1 1-66
nesting levels, for triggers Vol. 1 1-100
nonclustered indexes Vol. 1 1-51, Vol. 1 1-54
physical reads (**statistics io**) Vol. 1 1-299
place holders in a format string Vol. 1 1-254
rows in **count(*)** Vol. 1 2-2, Vol. 1 2-3
rows in **rowcnt** function Vol. 1 2-43, Vol. 1 2-46
scans (**statistics io**) Vol. 1 1-299
set textsize function Vol. 1 3-123, Vol. 1 3-148
stored procedure parameters Vol. 1 1-63
Sundays Vol. 1 2-21
tables allowed in a query Vol. 1 1-283, Vol. 1 2-4, Vol. 1 3-79
tables per database Vol. 1 1-82
timestamp columns Vol. 1 3-39
updates Vol. 1 1-101
work tables allowed Vol. 1 2-4

Number of characters
date interpretation and Vol. 1 3-38
in a column Vol. 1 3-29 to 3-30

Number of columns
in an **order by** clause Vol. 1 1-249
per table Vol. 1 1-15, Vol. 1 1-82
in a view Vol. 1 1-105

Number of pages
allocated to table or index Vol. 1 2-43
in an extent Vol. 1 1-55, Vol. 1 1-82
reserved_pgs function Vol. 1 2-43
statistics io and Vol. 1 1-299
used by table and clustered index (total) Vol. 1 2-43
used by table or index Vol. 1 2-41, B-26
used_pgs function Vol. 1 2-43
written (**statistics io**) Vol. 1 1-299

Numbers
See also Code numbers; IDs, user
asterisks (**) for overlength Vol. 1 2-38
converting strings of Vol. 1 3-29
database ID Vol. 1 2-41
datatype code Vol. 2 2-3
device Vol. 2 1-181
error return values (Server) Vol. 1 1-267
global variable unit Vol. 2 1-228
in mathematical function expressions Vol. 1 2-24
message Vol. 2 1-23, Vol. 2 1-144, Vol. 2 1-169
ODBC datatype code Vol. 2 2-3
odd or even binary Vol. 1 3-33
place holder (%*nn*!) Vol. 1 1-253
procid setting Vol. 1 1-298
random float Vol. 1 2-26
same name group procedure Vol. 1 1-58, Vol. 1 1-152, Vol. 1 1-187
select list Vol. 1 1-286
statistics io Vol. 1 1-299
virtual device Vol. 1 1-130, Vol. 1 1-133, Vol. 1 1-138
weekday names and Vol. 1 1-296, Vol. 1 2-22, Vol. 2 1-15

Numeric datatypes and row aggregates Vol. 1 2-29
Numeric expressions Vol. 1 xxii, Vol. 2 xix
round function for Vol. 1 2-26

nvarchar datatype **Vol. 1 3-29** to **3-30**
spaces in Vol. 1 3-29

O

Object Allocation Map (OAM) Vol. 1 1-111
Object names, database
See also Identifiers
as parameters Vol. 1 1-59
checking with **sp_checknames** Vol. 2 1-83
checking with **sp_checkreswords** Vol. 2 1-88
in stored procedures Vol. 1 1-64, Vol. 1 1-66
user-defined datatype names as Vol. 1 3-40
Object owners. *See* Database object owners
Object permissions
See also Command permissions; Permissions
grant Vol. 1 1-196 to 1-206
grant all Vol. 1 1-204
Object. *See* Database objects
object_id system function Vol. 1 2-42
object_name system function Vol. 1 2-42, Vol. 2 1-213
Objects referencing, **create procedure** and Vol. 1 1-63
Objects. *See* Database objects; Databases
ODBC. *See* Open Database Connectivity (ODBC) API
of option, **declare cursor** Vol. 1 1-119, Vol. 1 3-14
Official language name Vol. 2 1-16, Vol. 2 1-261
See also Aliases; Languages, alternate
Offset position
column B-13
readtext command Vol. 1 1-261
offsets option, **set** Vol. 1 1-297
on keyword
alter database Vol. 1 1-6
alter table Vol. 1 1-13
create index Vol. 1 1-54, Vol. 1 1-55
create table Vol. 1 1-79, Vol. 1 1-80
Open Client DB-Library applications
keywords Vol. 1 1-297
procid setting Vol. 1 1-298
set options for Vol. 1 1-297, Vol. 1 1-303
open command **Vol. 1 1-246** to **1-247**
Open Database Connectivity (ODBC) API datatypes Vol. 2 2-3
OpenVMS systems
contiguous option on Vol. 1 1-134, Vol. 1 3-48
mirroring options Vol. 1 1-135
Operator Vol. 1 3-100
assigning role Vol. 2 1-256
Operators
arithmetic Vol. 1 3-53
bitwise Vol. 1 3-54 to 3-55
comparison Vol. 1 3-56
precedence Vol. 1 3-53
Optimization of queries, **sp_recompile** Vol. 2 1-240
optimized report
dbcc indexalloc Vol. 1 1-112
dbcc tablealloc Vol. 1 1-111
Options
displaying / changing Vol. 2 1-115 to 1-121
or keyword
in expressions Vol. 1 3-58
in joins Vol. 1 3-80
search conditions Vol. 1 3-109
where Vol. 1 1-333
Order
See also Indexes; Precedence; Sort order
of arguments in translated strings Vol. 1 1-253
ascending sort Vol. 1 1-248, Vol. 1 1-286

of column list and insert data Vol. 1 1-219
of columns (fixed and variable length) Vol. 1 1-250
columns and row aggregates Vol. 1 1-35, Vol. 1 2-31
of creating indexes Vol. 1 1-55
of date parts Vol. 1 1-296, Vol. 2 1-15
descending Vol. 1 1-248
descending sort Vol. 1 1-286
error message arguments Vol. 1 1-253
of evaluation Vol. 1 1-314
of execution of operators in expressions Vol. 1 3-53
of names in a group Vol. 1 1-215
of null values Vol. 1 1-249, Vol. 1 3-94
of parameters in **create procedure** Vol. 1 1-188, Vol. 1 1-189
reversing character expression Vol. 1 2-35
for unbinding a rule Vol. 1 1-69
weekday numeric Vol. 1 2-22
order by clause **Vol. 1 1-248** to **1-251**
compute by and Vol. 1 1-35, Vol. 1 1-249, Vol. 1 1-286
not allowed in subqueries Vol. 1 3-110
select Vol. 1 1-286
Order of commands Vol. 1 1-202, Vol. 1 1-273
Original identity, resuming an. *See* **setuser** command
Other users, qualifying objects owned by Vol. 1 3-65
Outer joins **Vol. 1 3-82**
views and Vol. 1 1-106
Outer queries. *See* Subqueries
Output
See also Results; Variables
dbcc messages or errors Vol. 1 1-114
zero-length string Vol. 1 1-255
output option
create procedure Vol. 1 1-59, Vol. 1 1-188, Vol. 1 1-189
execute Vol. 1 1-188
return parameter Vol. 1 1-188, Vol. 1 3-97
sp_getmessage Vol. 2 1-169
Output packets, number of Vol. 1 3-147, Vol. 2 1-228
Overflow errors
datatypes Vol. 1 3-45
DB-Library Vol. 1 2-7, Vol. 1 2-31, Vol. 1 2-32
set arithabort and Vol. 1 1-294
Overhead
triggers Vol. 1 1-98
Override. *See* **with override** option
Overwriting triggers Vol. 1 1-97
Owners. *See* Database object owners; Database Owners
Ownership
See also Permissions; **setuser** command
of command and object permissions Vol. 1 1-199
dump devices and Vol. 2 1-47
of objects being referenced Vol. 1 3-65
of rules Vol. 1 1-70
of stored procedures Vol. 1 1-64, Vol. 1 1-66

P

@@pack_received global variable Vol. 1 3-147
sp_monitor and Vol. 2 1-228
@@pack_sent global variable Vol. 1 3-147
sp_monitor and Vol. 2 1-228
@@packet_errors global variable Vol. 1 3-147
sp_monitor and Vol. 2 1-228
Padding, data
See also Truncation
blanks and Vol. 1 1-221, Vol. 1 3-29
image datatype Vol. 1 3-123
null values and Vol. 1 3-89
underscores in temporary table names Vol. 1 3-61, Vol. 1 3-116
with zeroes Vol. 1 3-32

Page locks
types of Vol. 2 1-213
Pages (data)
See also Index pages; Table pages
allocation of Vol. 1 2-43
chain of Vol. 1 3-120
computing number of, with **sp_spaceused** Vol. 2 1-263
data_pgs system function Vol. 1 2-41, Vol. 1 2-46
in an extent Vol. 1 1-55, Vol. 1 1-82
locks held on Vol. 2 1-213
multibyte characters and Vol. 1 1-113
not allocated in the extent Vol. 1 1-111
reserved_pgs system function Vol. 1 2-43
statistics io and Vol. 1 1-299
truncate table and Vol. 1 1-311
used for internal structures Vol. 1 2-41, Vol. 1 2-43
used in a table or index Vol. 1 2-41, Vol. 1 2-43
used_pgs system function Vol. 1 2-43, Vol. 1 2-46
Pair of columns. *See* Common keys; Joins
Pair, mirrored Vol. 1 1-143
Parameters, configuration. *See* Configuration variables
Parameters, **create procedure**
datatypes Vol. 1 1-59
defaults Vol. 1 1-59
names Vol. 1 1-58 to 1-59
Parameters, procedure **Vol. 1 3-96** to **3-98**
See also Local variables
execute and Vol. 1 1-188
not part of transactions Vol. 1 1-190
ways to supply Vol. 1 1-188, Vol. 1 1-189
Parentheses ()
See also Symbols section of this index
in an expression Vol. 1 2-19, Vol. 1 3-59
in SQL statements Vol. 1 xix, Vol. 2 xvii
in user-defined datatypes Vol. 2 1-40
parseonly option, **set** Vol. 1 1-298
Partial characters, reading Vol. 1 1-262
Passwords Vol. 1 3-86, Vol. 2 1-20
date of last change Vol. 2 1-129
encryption Vol. 2 1-259
sp_password Vol. 2 1-230 to 1-232
sp_remoteoption and Vol. 2 1-245
sp_serveroption and Vol. 2 1-259
trusted logins or verifying Vol. 2 1-245
Path name
hard-coded or logical device Vol. 1 1-132
mirror device Vol. 1 1-134, Vol. 1 3-48
patindex string function Vol. 1 2-35, Vol. 1 2-37
See also Wildcard characters
text/image function Vol. 1 2-48, Vol. 1 3-123
Pattern matching
See also **patindex**; String functions; Wildcard characters
charindex string function Vol. 1 2-34
difference string function Vol. 1 2-34, Vol. 1 2-39
patindex string function Vol. 1 2-35, Vol. 1 2-48
wildcard Vol. 2 2-2
PC DB-Library. *See* DB-Library programs
Percent sign (%)
in messages Vol. 1 1-253
literal in error messages Vol. 1 1-255
modulo operator Vol. 1 3-53
place holder in error message Vol. 1 1-253
wildcard Vol. 1 3-57, Vol. 1 3-106
Performance Vol. 1 1-264
select into and Vol. 1 1-291
showplan and diagnostics Vol. 1 1-299
triggers and Vol. 1 1-98
writetext during **dump database** Vol. 1 1-339

Period (.) separator for qualifier names Vol. 1 3-63
Periods (..) for omitted name elements Vol. 1 3-63
Permissions
  assigned by Database Owner Vol. 1 1-196
  assigning Vol. 1 1-196
  changing with **setuser** Vol. 1 1-306
  command **Vol. 1 1-200** to **1-202**
  creating with **create schema** Vol. 1 1-72 to 1-74
  displaying user's Vol. 2 1-129
  dump devices and Vol. 2 1-47
  errors Vol. 1 3-139
  grant Vol. 1 1-196 to 1-206
  granting Vol. 2 1-195
  groups and Vol. 1 1-273
  information on Vol. 2 1-195
  new Database Owner Vol. 2 1-79
  new database user Vol. 2 1-221
  object Vol. 1 1-201
  "public" group Vol. 1 1-200 to 1-202
  readtext and column Vol. 1 3-93
  revoke command Vol. 1 1-269 to 1-274
  revoking Vol. 2 1-195
  sp_column_privileges information on Vol. 2 2-4 to 2-7
  sysprotects table B-45
  system procedures Vol. 2 1-6
  system tables B-3
  system tables entries for B-45
  writetext and column Vol. 1 3-93
Phantoms in transactions Vol. 1 3-133
Physical database consistency. *See* **dbcc** (Database Consistency Checker)
Physical datatypes Vol. 2 1-40
Physical device name Vol. 2 1-46
Physical reads (**statistics io**) Vol. 1 1-299
physname option
  disk init Vol. 1 1-130
  disk init, in OpenVMS Vol. 1 1-132
  disk reinit Vol. 1 1-138
pi mathematical function **Vol. 1 2-26**
Place holders
  print message Vol. 1 1-253
Plan
  create procedure and Vol. 1 1-60
  object B-42
Platform dependence, binary datatypes and Vol. 1 3-33
Plus (**+**)
  arithmetic operator Vol. 1 3-53
  concatenation operator Vol. 1 2-33
  string function Vol. 1 2-33
Pointers, device. *See* Segments
Pointers, *text* or *image* column Vol. 1 1-261, Vol. 1 2-48, Vol. 1 2-49, Vol. 1 3-120, Vol. 1 3-123
Positioning cursors Vol. 1 3-14
Pound sign (#)
  temporary table name prefix Vol. 1 1-75, Vol. 1 3-117
power mathematical function **Vol. 1 2-26**
Precedence
  See also Hierarchy
  of column order over order of aggregates Vol. 1 2-31
  default binding Vol. 1 1-47
  of lower and higher datatypes Vol. 1 3-59
  of operators in expressions Vol. 1 3-53
  order-sensitive commands and Vol. 1 1-202, Vol. 1 1-273
  rule binding Vol. 1 1-70, Vol. 2 1-76
  of user-defined return values Vol. 1 1-268
Preceding blanks. *See* Blanks; Spaces, character
Precision, datatype (monetary types) Vol. 1 3-34
Pre-defined global variables (@@) Vol. 1 3-145
Preference, uppercase letter sort order Vol. 1 1-250
Prefix, *locktype* information Vol. 2 1-213
prepare transaction command **Vol. 1 1-252**

primary key constraint
alter table Vol. 1 1-12
create table Vol. 1 1-78
Primary keys Vol. 1 1-82
pubs2 database C-18
sp_dropkey procedure Vol. 2 1-137
sp_foreignkey and Vol. 2 1-166
sp_helpkey and Vol. 2 1-188
sp_pkeys information on Vol. 2 2-17 to 2-18
sp_primarykey definition of Vol. 2 1-235
syskeys table B-29
primary option, **disk unmirror** Vol. 1 1-143
print command **Vol. 1 1-253** to **1-255**
local variables and Vol. 1 1-118
using **raiserror** or Vol. 1 1-255
Printing user-defined messages Vol. 1 1-253 to 1-255
Privileges. *See* Permissions
Probe Process, Two Phase Commit B-36
proc_role system function Vol. 1 2-43, Vol. 1 2-46
Procedure calls. *See* Remote procedure calls
Procedure groups Vol. 1 1-58, Vol. 1 1-152, Vol. 1 1-187
Procedure plan, **create procedure** and Vol. 1 1-60
Procedures. *See* Stored procedures; System procedures
Process logical name. *See* Logical device name
Processes (Server tasks)
See also Servers
checking locks on Vol. 2 1-212 to 1-214
ID number Vol. 1 1-227, Vol. 2 1-286
infected Vol. 2 1-287
infected, **waitfor errorexit** Vol. 1 1-328
killing Vol. 1 1-227 to 1-229
sp_who report on Vol. 1 1-227, Vol. 2 1-286 to 1-287
sysprocesses table B-43
system tables entries for B-43
processexit keyword
waitfor Vol. 1 1-327
@@procid global variable Vol. 1 3-147
procid option, **set** Vol. 1 1-298
Prompts, **sp_volchanged** Vol. 2 1-282 to 1-285
Protection system
command and object permissions Vol. 1 1-199
"public" group Vol. 1 1-205, Vol. 1 1-270, Vol. 1 1-273, B-59
See also Groups
grant and Vol. 1 1-197
information report Vol. 2 1-182
permissions Vol. 1 1-200 to 1-202
sp_addgroup and Vol. 2 1-13
sp_adduser and Vol. 2 1-49
sp_changegroup and Vol. 2 1-81
sp_dropgroup and Vol. 2 1-135
sp_helpgroup report on Vol. 2 1-182
public keyword
grant Vol. 1 1-197
revoke Vol. 1 1-270
publishers table, *pubs2* database C-1
pubs2 database **C-1** to **C-20**
defaults C-19
diagram C-20
objects C-19
organization chart C-20
primary and foreign keys C-18
rules C-19
table names C-1
view C-19
Punctuation
characters allowed in identifiers Vol. 1 3-61
enclosing in quotation marks Vol. 2 1-7
in parameter values Vol. 1 3-98
in user-defined datatypes Vol. 2 1-40

Q

qq. *See* **quarter** date part

Qualifier names Vol. 1 3-63, Vol. 1 3-65
quarter date part Vol. 1 2-20
Queries
compilation and optimization Vol. 2 1-240
compilation without execution Vol. 1 1-297, Vol. 1 1-298
execution settings Vol. 1 1-293 to 1-305
keywords list Vol. 1 1-297
nesting subqueries Vol. 1 3-110 to 3-115
showplan setting Vol. 1 1-299
sp_tables and Vol. 2 2-35
syntax check (**set parseonly**) Vol. 1 1-298
trigger firing per Vol. 1 1-98
union Vol. 1 1-313 to 1-316
views and Vol. 1 1-106
with / without **group by** and **having** Vol. 1 1-210
Question marks (??)
for partial characters Vol. 1 1-262
Quotation marks (" ")
comparison operators and Vol. 1 3-56
for empty strings Vol. 1 3-59, Vol. 1 3-93
enclosing constant values Vol. 1 2-37
enclosing *datetime* values Vol. 1 3-35
enclosing punctuation Vol. 1 3-98
enclosing reserved words Vol. 2 1-90
enclosing values in Vol. 2 1-7
in expressions Vol. 1 3-60
literal specification of Vol. 1 3-60
single, and **quoted_identifier** Vol. 2 1-98
Quoted identifiers
testing Vol. 2 1-90
using Vol. 2 1-89, Vol. 2 1-97 to 1-98
quoted_identifier option, **set** Vol. 1 1-298

R

radians mathematical function Vol. 1 2-26
Radians, conversion to degrees Vol. 1 2-25
raiserror command **Vol. 1 1-256** to **1-260**
compared to **print** Vol. 1 1-259
local variables and Vol. 1 1-118
using **print** or Vol. 1 1-255
rand mathematical function **Vol. 1 2-26**, Vol. 1 2-28
Range
See also Numbers; Size
of date parts values Vol. 1 2-20
datediff results Vol. 1 2-22
errors in mathematical functions Vol. 1 2-27
of monetary values allowed Vol. 1 3-34
of recognized dates Vol. 1 3-34
in search conditions Vol. 1 3-107
set rowcount Vol. 1 1-298
wildcard characters specifying Vol. 1 3-58, Vol. 1 3-151
Range-end keyword, **and** Vol. 1 3-57, Vol. 1 3-107
Range-start keyword, **between** Vol. 1 3-57, Vol. 1 3-107
read only database option
setting with **sp_dboption** Vol. 2 1-119
Read-only cursors Vol. 1 1-122, Vol. 1 3-14
Reads
dirty Vol. 1 3-132
non-repeatable Vol. 1 3-133
readtext command **Vol. 1 1-261** to **1-263**
text data initialization requirement Vol. 1 3-122
real datatype **Vol. 1 3-28**
Rebuilding
automatic, of nonclustered index Vol. 1 1-55
indexes Vol. 1 1-113
system tables Vol. 1 1-112
Recompilation
create procedure with recompile option Vol. 1 1-60, Vol. 1 1-63
dependent objects definition and Vol. 2 1-249

execute with recompile option Vol. 1 1-188
stored procedures Vol. 1 1-63, Vol. 2 1-240 to 1-241
without notice Vol. 2 1-249
reconfigure command **Vol. 1 1-264** to **1-265**
sp_configure and Vol. 2 1-106
reconfigure with override option
sp_configure Vol. 2 1-106
Records, audit Vol. 1 3-4, Vol. 2 1-11
Recovery
dump transaction and Vol. 1 1-182
time and **checkpoint** Vol. 1 1-25
time and transaction size Vol. 1 3-128
Recovery of *master* database Vol. 1 1-169
after using **create database** Vol. 1 1-43
after using **disk init** Vol. 1 1-132
Recreating
indexes Vol. 1 1-113
procedures Vol. 1 1-64
tables Vol. 1 1-156
Recursions, limited Vol. 1 1-101
Reference information
catalog stored procedures Vol. 2 2-1
system procedures Vol. 2 1-1 to 1-8
Transact-SQL commands Vol. 1 1-1 to 1-5
Transact-SQL functions Vol. 1 2-1
Transact-SQL topics Vol. 1 3-1 to 3-2
references constraint
alter table Vol. 1 1-13
create table Vol. 1 1-79
Referencing, object. *See* Dependencies, database object
Referential integrity constraints Vol. 1 1-88
binding user messages to Vol. 2 1-72
create table and Vol. 1 1-85
cross-database Vol. 1 1-90, Vol. 1 1-157
renaming Vol. 2 1-248 to 1-250
sp_indsuspect system procedure Vol. 2 1-210
sysconstraints table B-18
sysobjects table B-40 to B-41
sysreferences table B-47
triggers and Vol. 1 1-94 to 1-102
reindex option, **dbcc** Vol. 1 1-112 to 1-113
after **sp_indsuspect** Vol. 2 1-210
Reinitializing, **disk reinit** and Vol. 1 1-138 to 1-139
Relational expressions Vol. 1 3-52
See also Comparison operators
Remapping database objects Vol. 2 1-242 to 1-244
Remarks text. *See* Comments
Remirroring. *See* Disk mirroring
Remote logins
See also Logins; Users
dropping Vol. 2 1-146 to 1-147
information on Vol. 2 1-193
sp_remoteoption for Vol. 2 1-245 to 1-247
sysremotelogins table B-49, Vol. 2 1-25 to 1-27
system tables entries for B-49
trusted or untrusted mode Vol. 2 1-245
Remote procedure calls Vol. 1 1-289
auditing Vol. 1 3-3, Vol. 2 1-62
execute and Vol. 1 1-190
rollback and Vol. 1 1-276
sp_password Vol. 2 1-231
sysremotelogins table and B-49
sysservers table and B-52
user-defined transactions Vol. 1 3-128, Vol. 1 3-141
Remote servers Vol. 1 1-289
See also Servers
changing names of Vol. 2 1-94, Vol. 2 1-96
information on Vol. 2 1-200
information on logins of Vol. 2 1-193
passwords on Vol. 2 1-231
sp_dropremotelogin and Vol. 2 1-146
sp_remoteoption and Vol. 2 1-245 to 1-247
sysservers table B-52
system tables entries for B-52

Remote users. *See* Remote logins
remove option, **disk unmirror** Vol. 1 1-143, Vol. 1 3-49
Removing. *See* Dropping
Renaming **Vol. 2 1-248** to **1-250**
See also **sp_rename** system procedure
a database Vol. 2 1-251 to 1-253
identity of object owner Vol. 1 1-199
stored procedures Vol. 1 1-63
triggers Vol. 1 1-98
views Vol. 1 1-106
warnings about Vol. 2 1-249, Vol. 2 1-252
Repairing a damaged database Vol. 1 1-112
Repeated execution. *See* **while** loop
Repeating subquery. *See* Subqueries
replace keyword, **alter table** Vol. 1 1-14
replicate string function Vol. 1 2-35
Reports
sp_who Vol. 1 1-227, Vol. 2 1-286 to 1-287
types of **dbcc** Vol. 1 1-111
Reserved columns B-3
Reserved connections. *See* **user connections** configuration variable
Reserved return status values Vol. 1 1-267
Reserved words **A-1** to **A-6**
See also Keywords
APT-SQL A-3
database object identifiers and Vol. 1 3-61
as identifiers Vol. 2 1-85 to 1-99
SQL92 A-4 to A-5
system procedures and Vol. 2 1-7
Transact-SQL A-1 to A-2
reserved_pgs system function Vol. 1 2-43
Response time. *See* **waitfor** command
Restarting **while** loops Vol. 1 1-39
Restarts, Server Vol. 1 1-264
after using **disk refit** Vol. 1 1-137
after using **sp_dropdevice** Vol. 2 1-133
before using **create database** Vol. 1 1-41
rowcnt and Vol. 1 2-47
using **dataserver** utility Vol. 1 1-136, Vol. 1 1-141
Restoring
See also Recovery
a damaged *master* database Vol. 1 1-137, Vol. 1 1-138
database with **load database** Vol. 1 1-230 to 1-237
Restrictions
text and *image* columns Vol. 1 2-49
Result set
cursors Vol. 1 1-122, Vol. 1 3-14
Results
See also Output
of aggregate operations Vol. 1 1-209
null value operations and Vol. 1 3-88 to 3-95
order by and sorting **Vol. 1 1-248** to **1-251**
of row aggregate operations Vol. 1 2-29
retain option, **disk unmirror** Vol. 1 1-143
Retrieving
See also Search conditions; **select** command
current date and time Vol. 1 2-19
error message text Vol. 1 1-253, Vol. 2 1-169
null values Vol. 1 3-90
similar-sounding words or names Vol. 1 2-38
return command **Vol. 1 1-266** to **1-268**
Return parameters
output keyword Vol. 1 1-59, Vol. 1 1-188, Vol. 1 3-97
Return status
stored procedure Vol. 1 1-187, Vol. 1 1-266, Vol. 1 3-98
system procedures Vol. 2 1-7
Returning rows, cursor result set Vol. 1 1-192
reverse string function Vol. 1 2-35
revoke command **Vol. 1 1-269** to **1-274**

auditing use of Vol. 2 1-52
object and command permissions Vol. 1 1-200
sysprotects table B-45
Revoking roles with **sp_role** Vol. 2 1-256 to 1-257
right string function Vol. 1 2-35
role option, **set** Vol. 1 1-298
Roles **Vol. 1 3-99** to **3-104**
auditing commands requiring Vol. 2 1-62
auditing toggling of Vol. 2 1-62
commands requiring, list of Vol. 1 3-101 to 3-104
granting Vol. 1 1-198, Vol. 2 1-256 to 1-257
Operator Vol. 1 3-100
permissions and Vol. 1 1-205
proc_role system function Vol. 1 2-43, Vol. 1 2-46
revoking Vol. 2 1-256 to 1-257
show_role system function Vol. 1 2-43
stored procedures and Vol. 1 1-204
syslogin roles table B-35
sysroles table B-50
syssrvroles table B-53
System Administrator Vol. 1 3-99
System Security Officer Vol. 1 3-99
Roll back processes
checkpoint and Vol. 1 1-26
parameter values and Vol. 1 1-190
rollback command **Vol. 1 1-275** to **1-276**
See also Transactions
begin transaction and Vol. 1 1-21
commit and Vol. 1 1-28
in stored procedures Vol. 1 3-128
triggers and Vol. 1 1-99, Vol. 1 1-101, Vol. 1 3-128
rollback transaction command. *See* **rollback** command
rollback trigger command Vol. 1 1-99, **Vol. 1 1-277** to **1-278**
rollback work command. *See* **rollback** command
round mathematical function **Vol. 1 2-26**
Rounding Vol. 1 2-26
approximate numeric values Vol. 1 3-28
monetary values Vol. 1 3-34
str string function and Vol. 1 2-38
Row aggregates **Vol. 1 2-29** to **2-32**
compute and Vol. 1 1-30, Vol. 1 2-6
difference from aggregate functions Vol. 1 2-30
list of Vol. 1 2-29
rowcnt system function Vol. 1 2-43, **Vol. 1 2-46**
@@rowcount global variable Vol. 1 1-297, Vol. 1 3-147
cursors and Vol. 1 3-21
fetch and Vol. 1 1-194
triggers and Vol. 1 1-99
rowcount option, **set** Vol. 1 1-298
Rows, table
See also **select** command
aggregate functions applied to Vol. 1 1-209
comparison order of Vol. 1 1-250
computing number of, with **sp_spaceused** Vol. 2 1-263
create index and duplication of Vol. 1 1-50, Vol. 1 1-52
cursors Vol. 1 3-14 to 3-24
deleting with **truncate table** Vol. 1 1-311
detail and summary results Vol. 1 2-29 to 2-32
displaying command-affected Vol. 1 1-297
grouping Vol. 1 1-207
insert Vol. 1 1-220
number of Vol. 1 2-43, Vol. 1 2-46
row aggregates and Vol. 1 2-29 to 2-32
rowcount setting Vol. 1 1-298
scalar aggregates applied to Vol. 1 1-209
uniquely identifying Vol. 1 3-67
update Vol. 1 1-317 to 1-323
ways to group Vol. 1 1-209

roysched table, *pubs2* database C-14 to C-16
rtrim string function Vol. 1 2-35
Rules
See also Database objects
batch queries and Vol. 1 3-7
binding Vol. 1 1-68, Vol. 1 1-70, Vol. 2 1-75 to 1-78
changing names of Vol. 2 1-92
checking name with **sp_checkreswords** Vol. 2 1-88
column definition conflict with Vol. 1 1-70, Vol. 1 3-94
creating new Vol. 1 1-68 to 1-71
default violation of Vol. 1 1-47
displaying the text of Vol. 2 1-204
dropping user-defined Vol. 1 1-154
for finding objects Vol. 2 1-125, Vol. 2 1-173
for identifiers Vol. 1 3-61 to 3-66
insert and Vol. 1 1-221
naming user-created Vol. 1 1-68, Vol. 2 1-75
pubs2 database C-19
remapping Vol. 2 1-242 to 1-244
renaming Vol. 2 1-248 to 1-250
for sort order ties Vol. 1 1-250 to 1-251
system tables entries for B-15, B-40 to B-41, B-42
unbinding with **sp_unbindrule** Vol. 2 1-276 to 1-278
violations in user transaction Vol. 1 3-141
Running a procedure with **execute** Vol. 1 1-187 to 1-191

S

"sa" login
server user IDs and Vol. 1 2-46
sales table, *pubs2* database C-12
salesdetail table, *pubs2* database C-9 to C-11
save transaction command **Vol. 1 1-279** to **1-280**
See also Transactions
Savepoints
See also Checkpoint process
rollback and Vol. 1 1-275
setting using **save transaction** Vol. 1 1-279, Vol. 1 3-125
transactions Vol. 1 3-130
Scalar aggregates
group by and Vol. 1 1-209
nesting vector aggregates within Vol. 1 2-5
Scalar values, theta joins of Vol. 1 3-80
Scans, cursor Vol. 1 1-122, Vol. 1 3-19
Scans, number of (**statistics io**) Vol. 1 1-299
Schemas **Vol. 1 1-72** to **1-74**
permissions Vol. 1 1-73
Scope of cursors Vol. 1 1-121, Vol. 1 3-18
Search conditions **Vol. 1 3-105** to 3-109
See also **like** keyword; Retrieving
datetime data Vol. 1 3-38
group by and **having** query Vol. 1 1-208, Vol. 1 1-211, Vol. 1 3-105
select Vol. 1 1-284
where clause Vol. 1 1-329 to 1-335
second date part Vol. 1 2-20
secondary option, **disk unmirror** Vol. 1 1-143, Vol. 1 3-49
Seconds, **datediff** results in Vol. 1 2-22
Security
See also Permissions
command and object permissions Vol. 1 1-199
passwords Vol. 1 3-86
stored procedure ownership and Vol. 1 1-65
views and Vol. 1 1-105
Segments
See also Database devices; Log segment; Space allocation
adding Vol. 2 1-28 to 1-30
changing names of Vol. 2 1-94, Vol. 2 1-97

checking names with **sp_checkreswords** Vol. 2 1-89
creating indexes on Vol. 1 1-13, Vol. 1 1-54, Vol. 1 1-79
dbcc checktable report on Vol. 1 1-110
dbcc indexalloc report on Vol. 1 1-112
dropping Vol. 2 1-148 to 1-150
extending Vol. 2 1-28, Vol. 2 1-163
information on Vol. 2 1-198
last device reference for Vol. 2 1-150
mapping Vol. 2 1-29
mapping to a new device Vol. 1 1-9
monitoring remaining space Vol. 2 1-34 to 1-39, Vol. 2 1-222 to 1-226
names of Vol. 1 1-13, Vol. 1 1-54, Vol. 1 1-55, Vol. 1 1-79, Vol. 1 1-80, Vol. 2 1-29
number of named Vol. 1 1-43, Vol. 2 1-29
separation of table and index Vol. 1 1-55, Vol. 1 1-84
sp_helpthreshold report on Vol. 2 1-206
syssegments table B-51
system tables entries for B-51
select command **Vol. 1 1-281** to **1-292**
aggregates and Vol. 1 2-2, Vol. 1 2-4
auditing use of Vol. 2 1-58
create procedure and Vol. 1 1-63
create view and Vol. 1 1-103
with **distinct**, null values and Vol. 1 3-94
restrictions in standard SQL Vol. 1 2-5
in Transact-SQL compared to standard SQL Vol. 1 2-5
triggers and Vol. 1 1-97
union operation with Vol. 1 1-313
select into command Vol. 1 1-282 to 1-291
checkpoint and Vol. 1 1-25
column changes Vol. 1 1-15
IDENTITY columns and Vol. 1 3-68 to 3-70
not allowed with **compute** Vol. 1 1-34, Vol. 1 1-286, Vol. 1 2-31
temporary table Vol. 1 3-118
select into/bulkcopy database option Vol. 1 1-290
dump transaction and Vol. 1 1-180
Select list Vol. 1 1-282, Vol. 1 1-286
union statements Vol. 1 1-314
select option, **create view** Vol. 1 1-103
select statement Vol. 1 1-281 to 1-292
embedded Vol. 1 1-125
for browse Vol. 1 3-8
group by and **having** clauses Vol. 1 1-207
insert and Vol. 1 1-222
local variables and Vol. 1 1-118, Vol. 1 3-143
size of *text* data to be returned with Vol. 1 1-299
variables and Vol. 1 1-117, Vol. 1 3-144
self_recursion option, **set** Vol. 1 1-101, Vol. 1 1-298
Self-joins
definition of Vol. 1 3-82
Sentence order and numbered place holders Vol. 1 1-253
Separation, physical
of table and index segments Vol. 1 1-55, Vol. 1 1-84
of transaction log device Vol. 1 1-135, Vol. 1 1-141, Vol. 1 3-47
Sequence tree, object B-42
Sequence. *See* **order by** clause; Sort order
serial option, **disk mirror** Vol. 1 1-134, Vol. 1 3-48
Server aliases Vol. 2 1-31
Server cursors Vol. 1 3-17
Server information options. *See* Information (Server)
Server process ID number. *See* Processes (Server tasks)
Server restarts. *See* Restarts, Server
Server user name and ID
number -1 guest account Vol. 1 2-46
suser_id function Vol. 1 2-43
suser_name function for Vol. 1 2-43
@@servername global variable Vol. 1 3-147

Servers
See also Processes (Server tasks); Remote servers
See also Remote servers
adding Vol. 2 1-31 to 1-33
attribute names Vol. 2 2-19 to 2-22
capacity for databases Vol. 1 1-42
dropping Vol. 2 1-151 to 1-152
information on remote Vol. 2 1-200
information on remote logins Vol. 2 1-193
local Vol. 2 1-31
names of Vol. 2 1-31
options, changing with **sp_serveroption** Vol. 2 1-258 to 1-260
sp_monitor statistics Vol. 2 1-227
sp_server_info information on Vol. 2 2-19 to 2-22
upgrading and **sp_checknames** Vol. 2 1-83
upgrading and **sp_checkreswords** Vol. 2 1-88
set command **Vol. 1 1-293** to **1-305**
See also individual **set** *options*
chained transaction mode Vol. 1 3-132
defaults Vol. 1 1-302
inside a stored procedure Vol. 1 1-65
inside a trigger Vol. 1 1-98
sp_setlangalias and **language** option Vol. 2 1-261
within **update** Vol. 1 1-317
set textsize function Vol. 1 3-123
Settable options. *See* Database options
setuser command **Vol. 1 1-306** to **1-307**
user impersonation using Vol. 1 1-199
7-bit terminal, **sp_helpsort** output Vol. 2 1-202
Severity levels, error
and user-defined messages Vol. 1 1-259
shared keyword
cursors and Vol. 1 3-23
select Vol. 1 1-284
Shared locks Vol. 2 1-213
show_role system function Vol. 1 2-43
showplan option, **set** Vol. 1 1-299
shutdown command **Vol. 1 1-308** to **1-310**
side option, **disk unmirror** Vol. 1 1-143, Vol. 1 3-49
sign mathematical function **Vol. 1 2-26**
Similar-sounding words. *See* **soundex** string function
sin mathematical function **Vol. 1 2-26**
Sine angle, mathematical function Vol. 1 2-25
Single character wildcards Vol. 1 3-58
Single quotes. *See* Quotation marks
single user database option
setting with **sp_dboption** Vol. 2 1-119
Single-byte character sets
char datatype for Vol. 1 3-29
Single-user mode
sp_renamedb and Vol. 2 1-251
Size
See also Length; Number (quantity of); Range; Size limit; Space allocation
@@textsize global variable Vol. 1 3-148
ceiling mathematical function Vol. 1 2-25
columns in table Vol. 1 1-15
compiled stored procedure Vol. 1 1-63
composite index Vol. 1 1-51
database device Vol. 1 1-131
database extension Vol. 1 1-6
estimation of a compiled stored procedure Vol. 1 1-63
floor mathematical function Vol. 1 2-25
identifiers (length) Vol. 1 3-61
image datatype Vol. 1 3-120
indexes Vol. 1 2-46
initialized database device Vol. 1 1-133
log device Vol. 1 1-131, Vol. 1 1-133, Vol. 2 1-218
model database Vol. 1 1-131
new database Vol. 1 1-41
of columns Vol. 1 2-41
of **pi** Vol. 1 2-26

readtext data Vol. 1 1-261, Vol. 1 1-262
recompiled stored procedures Vol. 1 1-63
row Vol. 1 1-15, Vol. 1 1-82
set textsize function Vol. 1 1-299, Vol. 1 3-123
tables Vol. 1 1-82, Vol. 1 2-46
text and *image* data Vol. 1 1-339
text data to be returned with **select** Vol. 1 1-299
text datatype Vol. 1 3-120
@@textsize global variable Vol. 1 3-148
transaction log device Vol. 1 1-44, Vol. 1 1-133
transaction logs Vol. 1 2-46
Size limit
columns allowed per table Vol. 1 1-82
integer value smallest or largest Vol. 1 2-25
of monetary values Vol. 1 3-34
print command Vol. 1 1-254
tables per database Vol. 1 1-82
Size of columns
approximate numeric datatype Vol. 1 3-28
character Vol. 1 3-29 to 3-30
by datatype Vol. 1 3-25 to 3-26
datetime datatype Vol. 1 3-34
fixed-length Vol. 1 3-29 to 3-30
image datatype Vol. 1 3-32
monetary values Vol. 1 3-34
smalldatetime datatype Vol. 1 3-35
text datatype Vol. 1 3-30
variable-length Vol. 1 3-29 to 3-30
size option
disk init Vol. 1 1-130
disk reinit Vol. 1 1-138
skip_ncindex option
dbcc checkdb Vol. 1 1-111
dbcc checktable Vol. 1 1-111
Slash (/) division operator Vol. 1 3-53
smalldatetime datatype **Vol. 1 3-34** to **3-35**
date functions an Vol. 1 2-20
smallint datatype **Vol. 1 3-27**
smallmoney datatype **Vol. 1 3-34**
Sort order
See also Order
ascending or descending Vol. 1 1-248
changes to default Vol. 1 1-265
choices and effects Vol. 1 1-249
comparison operators and Vol. 1 3-56
group by and **having** and Vol. 1 1-215
groups of names Vol. 1 1-215
reindex check after change Vol. 1 1-113
sp_helpsort display Vol. 2 1-202
syscharsets system table B-11
sorted_data option, **create index** Vol. 1 1-53
soundex string function Vol. 1 2-35, Vol. 1 2-38
sp_addalias system procedure Vol. 1 1-326, **Vol. 2 1-9** to **1-10**
sp_addauditrecord system procedure **Vol. 2 1-11** to **1-12**
sp_addgroup system procedure **Vol. 2 1-13** to **1-14**
sp_addlanguage system procedure **Vol. 2 1-15** to **1-19**
sp_addlogin system procedure **Vol. 2 1-20** to **1-22**
sp_addmessage system procedure **Vol. 2 1-23** to **1-24**
sp_addremotelogin system procedure **Vol. 2 1-25** to **1-27**
sp_addsegment system procedure **Vol. 2 1-28** to **1-30**
sp_addserver system procedure **Vol. 2 1-31** to **1-33**
sp_addthreshold system procedure **Vol. 2 1-34** to **1-39**
sp_addtype system procedure **Vol. 2 1-40** to **1-45**
sp_addumpdevice system procedure **Vol. 2 1-46** to **1-48**
sp_adduser system procedure **Vol. 2 1-49** to **1-51**
sp_auditdatabase system procedure **Vol. 2 1-52** to **1-54**

sp_auditlogin system procedure **Vol. 2 1-55** to **1-57**
sp_auditobject system procedure **Vol. 2 1-58** to **1-60**
sp_auditoption system procedure **Vol. 2 1-61** to **1-64**
sp_auditsproc system procedure **Vol. 2 1-65** to **1-67**
sp_bindefault system procedure **Vol. 2 1-68** to **1-71**
create default and Vol. 1 1-47, Vol. 2 1-69
user-defined datatypes and Vol. 1 3-40
sp_bindmsg system procedure **Vol. 2 1-72** to **1-74**
sp_bindrule system procedure **Vol. 2 1-75** to **1-78**
create rule and Vol. 1 1-69
user-defined datatypes and Vol. 1 3-40
sp_changedbowner system procedure **Vol. 2 1-79** to **1-80**
sp_changegroup system procedure **Vol. 2 1-81** to **1-82**
sp_dropgroup and Vol. 2 1-135
sp_checknames system procedure **Vol. 2 1-83** to **1-84**
sp_checkreswords system procedure **Vol. 2 1-85** to **1-99**
sp_clearstats system procedure **Vol. 2 1-100** to **1-101**
sp_column_privileges catalog stored procedure **Vol. 2 2-4** to **2-7**
sp_columns catalog stored procedure **Vol. 2 2-8** to **2-9**
datatype code numbers Vol. 2 2-3
and **sp_datatype_info** Vol. 2 2-12
sp_commonkey system procedure **Vol. 2 1-102** to **1-104**
sp_configure system procedure **Vol. 2 1-105** to **1-110**
reconfigure and Vol. 1 1-264
sp_cursorinfo system procedure Vol. 1 3-22, **Vol. 2 1-111** to **1-114**
sp_databases catalog stored procedure **Vol. 2 2-10**
sp_datatype_info catalog stored procedure **Vol. 2 2-12** to **2-13**
sp_dboption system procedure **Vol. 2 1-115** to **1-121**
checkpoints and Vol. 1 1-26
load transaction and Vol. 1 1-242
transactions and Vol. 1 3-128
sp_dbremap system procedure **Vol. 2 1-122** to **1-123**
sp_depends system procedure Vol. 1 1-83, **Vol. 2 1-124** to **1-126**
sp_diskdefault system procedure **Vol. 2 1-127** to **1-128**
sp_displaylogin system procedure **Vol. 2 1-129** to **1-130**
sp_dropalias system procedure **Vol. 2 1-131** to **1-132**
sp_dropdevice system procedure **Vol. 2 1-133** to **1-134**
sp_dropgroup system procedure **Vol. 2 1-135** to **1-136**
See also **sp_changegroup**
sp_dropkey system procedure **Vol. 2 1-137** to **1-139**
sp_droplanguage system procedure **Vol. 2 1-140** to **1-141**
sp_droplogin system procedure **Vol. 2 1-142** to **1-143**
sp_dropmessage system procedure **Vol. 2 1-144** to **1-145**
sp_dropremotelogin system procedure **Vol. 2 1-146** to **1-147**
sp_dropsegment system procedure **Vol. 2 1-148** to **1-150**
sp_placeobject and Vol. 2 1-149
sp_dropserver system procedure **Vol. 2 1-151** to **1-152**
sp_dropthreshold system procedure **Vol. 2 1-153** to **1-154**
sp_droptype system procedure **Vol. 2 1-155** to **1-156**
sp_dropuser system procedure **Vol. 2 1-157** to **1-158**

sp_estspace system procedure **Vol. 2 1-159** to **1-162**
sp_extendsegment system procedure **Vol. 2 1-163** to **1-165**
sp_fkeys catalog stored procedure **Vol. 2 2-14** to **2-16**
sp_foreignkey system procedure **Vol. 2 1-166** to **1-168**
sp_getmessage system procedure **Vol. 2 1-169** to **1-170**
sp_help system procedure Vol. 1 3-41, **Vol. 2 1-171** to **1-174**
IDENTITY columns and Vol. 1 3-70
sp_helpconstraint system procedure **Vol. 2 1-175** to **1-177**
sp_helpdb system procedure **Vol. 2 1-178** to **1-179**
sp_helpdevice system procedure **Vol. 2 1-180** to **1-181**
sp_helpgroup system procedure **Vol. 2 1-182** to **1-183**
sp_helpindex system procedure **Vol. 2 1-184** to **1-185**
sp_helpjoins system procedure **Vol. 2 1-186** to **1-187**
sp_helpkey system procedure **Vol. 2 1-188** to **1-189**
sp_helplanguage system procedure **Vol. 2 1-190** to **1-191**
sp_helplog system procedure **Vol. 2 1-192**
sp_helpremotelogin system procedure **Vol. 2 1-193** to **1-194**
sp_helprotect system procedure Vol. 2 1-195 to 1-197
sp_helpsegment system procedure **Vol. 2 1-198** to **1-199**
sp_helpserver system procedure **Vol. 2 1-200** to **1-201**
sp_helpsort system procedure **Vol. 2 1-202** to **1-203**
sp_helptext system procedure **Vol. 2 1-204** to **1-205**
sp_helpthreshold system procedure **Vol. 2 1-206** to **1-207**
sp_helpuser system procedure **Vol. 2 1-208** to **1-209**
sp_indsuspect system procedure **Vol. 2 1-210** to **1-211**
sp_lock system procedure **Vol. 2 1-212** to **1-214**
sp_locklogin system procedure **Vol. 2 1-215** to **1-216**
sp_logdevice system procedure **Vol. 2 1-217** to **1-219**
log on extension to **create database** and Vol. 2 1-217
sp_modifylogin system procedure **Vol. 2 1-220** to **1-221**
sp_modifythreshold system procedure **Vol. 2 1-222** to **1-226**
sp_monitor system procedure **Vol. 2 1-227** to **1-229**
sp_password system procedure **Vol. 2 1-230** to **1-232**
sp_pkeys catalog stored procedure **Vol. 2 2-17** to **2-18**
sp_placeobject system procedure **Vol. 2 1-233** to **1-234**
sp_primarykey system procedure **Vol. 2 1-235** to **1-236**
sp_foreignkey and Vol. 2 1-166
sp_procxmode system procedure **Vol. 2 1-237** to **1-239**
sp_recompile system procedure **Vol. 2 1-240** to **1-241**
sp_remap system procedure Vol. 2 1-242 to 1-244
sp_remoteoption system procedure **Vol. 2 1-245** to **1-247**
sp_rename system procedure **Vol. 2 1-248** to **1-250**
sp_renamedb system procedure Vol. 2 1-93, **Vol. 2 1-251** to **1-253**
sp_reportstats system procedure **Vol. 2 1-254** to **1-255**
sp_role system procedure **Vol. 2 1-256** to **1-257**

sp_server_info catalog stored procedure **Vol. 2 2-19** to **2-22**
sp_tables and Vol. 2 2-36
sp_serveroption system procedure **Vol. 2 1-258** to **1-260**
sp_setlangalias system procedure **Vol. 2 1-261** to **1-262**
sp_spaceused system procedure **Vol. 2 1-263** to **1-264**
sp_special_columns catalog stored procedure **Vol. 2 2-23** to **2-25**
sp_sproc_columns catalog stored procedure **Vol. 2 2-26** to **2-27**
datatype code numbers Vol. 2 2-3
sp_statistics catalog stored procedure **Vol. 2 2-28** to **2-29**
sp_stored_procedures catalog stored procedure **Vol. 2 2-30** to **2-31**
sp_server_info information Vol. 2 2-21
sp_syntax system procedure **Vol. 2 1-265** to **1-267**
sp_table_privileges catalog stored procedure **Vol. 2 2-32**
sp_tables catalog stored procedure **Vol. 2 2-35** to **2-36**
sp_server_info information Vol. 2 2-21
sp_thresholdaction system procedure **Vol. 2 1-268** to **1-270**
threshold procedure Vol. 2 1-35, Vol. 2 1-223
sp_unbindefault system procedure Vol. 1 1-148, **Vol. 2 1-271** to **1-273**
sp_unbindmsg system procedure **Vol. 2 1-274** to **1-275**
sp_unbindrule system procedure **Vol. 2 1-276** to **1-278**
create rule and Vol. 1 1-69
drop rule and Vol. 1 1-154
sp_volchanged system procedure **Vol. 2 1-279** to **1-285**
messages Vol. 2 1-282 to 1-285
sp_who system procedure **Vol. 2 1-286** to **1-287**
Space
See also Size; Space allocation
for a clustered index Vol. 1 1-13, Vol. 1 1-52, Vol. 1 1-55, Vol. 1 1-78
database storage Vol. 1 1-6, Vol. 1 1-13, Vol. 1 1-52, Vol. 1 1-56, Vol. 1 1-78
dbcc checktable reporting free Vol. 1 1-110
estimating Vol. 2 1-159 to 1-162
extents Vol. 1 1-55, Vol. 1 1-82, Vol. 1 1-111
freeing with **truncate table** Vol. 1 1-311
for index pages Vol. 1 1-12, Vol. 1 1-51 to 1-52, Vol. 1 1-78
monitoring remaining with **sp_addthreshold** Vol. 2 1-34 to 1-39
monitoring remaining with **sp_modifythreshold** Vol. 2 1-222 to 1-226
new database Vol. 1 1-41
for recompiled stored procedures Vol. 1 1-63
retrieving inactive log Vol. 1 1-175
running out of Vol. 1 1-175
sp_spaceused procedure Vol. 2 1-263 to 1-264
for stored procedures Vol. 1 1-63
used on the log segment Vol. 1 1-110, Vol. 1 1-175
Space allocation
See also Database devices; Segments
future Vol. 2 1-233 to 1-234
log device Vol. 1 1-44, Vol. 2 1-218
pages Vol. 1 1-111
sp_placeobject procedure Vol. 2 1-233 to 1-234
system tables entries for B-57
sysusages table B-57
table Vol. 1 1-82, Vol. 1 1-111
space string function Vol. 1 2-35

Spaces, character
See also Blanks
in character datatypes Vol. 1 3-29 to 3-32
empty strings (" ") or (' ') as Vol. 1 3-59, Vol. 1 3-93
inserted in text strings Vol. 1 2-35
like *datetime* values and Vol. 1 3-38
not allowed in identifiers Vol. 1 3-61
update of Vol. 1 1-320
#spdbdesc temporary table Vol. 2 1-8
#spdevtab temporary table Vol. 2 1-8
Speed (Server)
of *binary* and *varbinary* datatype access Vol. 1 3-32
of **create database for load** Vol. 1 1-43
of **create index** with **sorted_data** Vol. 1 1-54
of **dump transaction** compared to **dump database** Vol. 1 1-182
execute Vol. 1 1-190
of recovery Vol. 1 3-128
of **truncate table** compared to **delete** Vol. 1 1-311
writetext compared to **dbwritetext** and **dbmoretext** Vol. 1 1-339
@@spid global variable Vol. 1 3-147
spid number. *See* Processes (Server tasks)
#spindtab temporary table Vol. 2 1-8
Splitting
table or index Vol. 2 1-233
spt_committab table Vol. 2 1-8
spt_datatype_info table Vol. 2 2-2
spt_datatype_info_ext table Vol. 2 2-2
spt_monitor table Vol. 2 1-8
#spt_server temporary table Vol. 2 1-8
spt_server_info table Vol. 2 2-2
spt_values table Vol. 2 1-8
SQL statements
create trigger Vol. 1 1-94
in **create procedure** Vol. 1 1-60
SQL. *See* Transact-SQL
SQLSTATE codes Vol. 1 D-1 to D-7
exceptions Vol. 1 D-2 to D-7
@@sqlstatus global variable Vol. 1 3-147
cursors and Vol. 1 3-21
fetch and Vol. 1 1-193
sqrt mathematical function **Vol. 1 2-26**
Square brackets [ ]
caret wildcard symbol [^] and Vol. 1 3-57, Vol. 1 3-151
in SQL statements Vol. 1 xix, Vol. 2 xvii
wildcard specifier Vol. 1 3-57
Square root mathematical function Vol. 1 2-26
ss. *See* **second** date part
Standard SQL, aggregate functions in Vol. 1 2-5
startserver utility command
disk mirror and Vol. 1 1-136
disk remirror and Vol. 1 1-141
Static configuration values Vol. 1 1-264
Statistical data summary, **sp_monitor** report Vol. 2 1-227 to 1-229
Statistics
returned by global variables Vol. 2 1-227
sp_clearstats procedure Vol. 2 1-100
sp_monitor Vol. 2 1-227
sp_reportstats Vol. 2 1-254 to 1-255
update statistics Vol. 1 1-324
statistics io option, **set** Vol. 1 1-299
statistics time option, **set** Vol. 1 1-299
Status
database device Vol. 2 1-127
stored procedures execution Vol. 1 1-190
Stopping a procedure. *See* **return** command
Storage management
text and *image* data Vol. 1 3-121
Stored procedure triggers. *See* Triggers
Stored procedures
See also Database objects; System procedures
alter table and Vol. 1 1-15
catalog Vol. 2 2-1 to 2-36

changing transaction modes with **sp_procxmode** Vol. 2 1-237 to 1-239
checking for roles in Vol. 1 2-46
creating Vol. 1 1-58 to 1-67
dropping Vol. 1 1-58, Vol. 1 1-152 to 1-153
execute Vol. 1 1-187 to 1-191
executing Vol. 1 1-187 to 1-191
execution status of Vol. 1 1-190
granting permission to roles on Vol. 1 2-46
grouping Vol. 1 1-58, Vol. 1 1-152, Vol. 1 1-187
ID numbers Vol. 1 1-298
isolation levels Vol. 1 3-137
nesting Vol. 1 1-63, Vol. 1 1-190
object dependencies and B-22, Vol. 2 1-124 to 1-126
parameter datatype declaration Vol. 1 3-42
parameters Vol. 1 3-96 to 3-98
parseonly not used with Vol. 1 1-298
permissions granted Vol. 1 1-197, Vol. 1 1-270
permissions revoked Vol. 1 1-273
procid option Vol. 1 1-298
remapping Vol. 2 1-242 to 1-244
renamed database and Vol. 2 1-252
renaming Vol. 1 1-63, Vol. 2 1-248 to 1-250
return status Vol. 1 1-187, Vol. 1 1-266, Vol. 1 3-98
rollback in Vol. 1 3-128
set commands in Vol. 1 1-293
sp_checkreswords and Vol. 2 1-89
sp_recompile and Vol. 2 1-240 to 1-241
sp_sproc_columns information on Vol. 2 2-26 to 2-27
sp_stored_procedures information on Vol. 2 2-30 to 2-31
storage maximums Vol. 1 1-63
system tables entries for B-15, B-40 to B-41, B-42
temporary tables and Vol. 1 3-117
transactions and Vol. 1 3-131, Vol. 1 3-134 to 3-139
stores table, *pubs2* database C-13
str string function Vol. 1 2-36, Vol. 1 2-37
String functions **Vol. 1 2-33** to **2-39**
See also text datatype
String message, **print** Vol. 1 1-253
string_rtruncation
insert and Vol. 1 1-221
update and Vol. 1 1-320
string_rtruncation option, **set** Vol. 1 1-299
Strings
concatenating Vol. 1 3-55
Structure
See also Order
clustered and **nonclustered** index Vol. 1 1-50 to 1-51
configuration B-19
stuff string function Vol. 1 2-36, Vol. 1 2-38
Style values, date representation Vol. 1 2-10
Subgroups, summary values for Vol. 1 1-30
Subqueries **Vol. 1 3-110** to **3-115**
See also Joins
any keyword and Vol. 1 3-57
correlated or repeating Vol. 1 3-115
exists keyword in Vol. 1 3-114
in expressions Vol. 1 3-57
joins as Vol. 1 3-82, Vol. 1 3-84
nesting Vol. 1 3-110 to 3-115
null values and Vol. 1 3-94
order by and Vol. 1 1-249
substring string function Vol. 1 2-36
Subtraction operator (-) Vol. 1 3-53
Suffix names
locktype information Vol. 2 1-213
temporary table name Vol. 1 3-116
sum aggregate function **Vol. 1 2-3**
as row aggregate Vol. 1 2-29
Summary values
aggregate functions and Vol. 1 2-2
generation with **compute** Vol. 1 1-30

Sundays, number value Vol. 1 2-21
suser_id system function Vol. 1 2-43
suser_name system function Vol. 1 2-43
Suspect indexes. *See* **reindex** option, **dbcc**
syb_identity keyword
IDENTITY columns and Vol. 1 3-68
select and Vol. 1 1-289
sybsecurity database Vol. 1 3-3
dropping Vol. 1 1-147
sybsyntax database Vol. 2 1-266
sybsystemprocs database
permissions and Vol. 2 1-6
Symbols
See also Wildcard characters; *Symbols section of this index*
arithmetic operator Vol. 1 3-53
comparison operator Vol. 1 3-56
in identifier names Vol. 1 3-61
join operator Vol. 1 3-80
matching character strings Vol. 1 3-57
money Vol. 1 3-61
SQL statement Vol. 2 xvii to xix
wildcards Vol. 1 3-57
Syntax
catalog stored procedures Vol. 2 2-2
check using **set parseonly** Vol. 1 1-298
checking for reserved words Vol. 2 1-88
conventions Vol. 1 xix to xxi, Vol. 2 xvii to xix
display procedure (**sp_syntax**) Vol. 2 1-265 to 1-267
sysalternates table **B-5**, Vol. 2 1-131
aliases Vol. 2 1-9
sysusers table and Vol. 2 1-9
sysauditoptions table Vol. 1 3-3, **B-6** to **B-7**
sysaudits table Vol. 1 3-3, **B-8** to **B-10**
syscharsets table **B-11** to **B-12**
syscolumns table Vol. 1 1-112, Vol. 1 3-39, **B-13** to **B-14**
syscomments table **B-15**
default definitions in Vol. 1 1-47
text storage in Vol. 2 1-204
sysconfigures table **B-16** to **B-17**
database size parameter Vol. 1 1-43
reconfigure and Vol. 1 1-264
sysconstraints table **B-18**
sp_bindmsg and Vol. 2 1-72
syscurconfigs table **B-19**
sysdatabases table **B-20** to **B-21**, Vol. 2 2-10
sysdepends table **B-22**
sysdevices table **B-23** to **B-24**, Vol. 2 1-127, Vol. 2 1-180
disk init and Vol. 1 1-131
mirrornames column in Vol. 1 1-143
sysengines table **B-25**
sysindexes table **B-26** to **B-28**
composite indexes and Vol. 1 1-56
name column in Vol. 1 3-121
syskeys table **B-29** to **B-30**
sp_dropkey and Vol. 2 1-137
sp_foreignkey and Vol. 2 1-166
sp_primarykey and Vol. 2 1-235
syslanguages table **B-31** to **B-32**, Vol. 2 1-190
sp_droplanguages and Vol. 2 1-140
syslocks table **B-33** to **B-34**
sysloginroles table **B-35**
syslogins table **B-36** to **B-37**
sp_modifylogin and Vol. 2 1-221
syslogs table **B-38**, Vol. 2 1-217
See also Recovery; Transaction logs
danger of changing the B-3
infinite loop if changes to B-38
put on a separate device Vol. 1 1-135, Vol. 1 1-141, Vol. 1 3-47, Vol. 2 1-217
running **dbcc checktable** on Vol. 1 1-110
sysmessages table **B-39**
error message text Vol. 2 1-169
raiserror and Vol. 1 1-256
sysname custom datatype **Vol. 1 3-40**
sysobjects table **B-40** to **B-41**
sysprocedures table **B-42**
triggers in Vol. 1 1-97
sysprocesses table **B-43** to **B-44**

sysprotects table **B-45** to **B-46**
grant/revoke statements and Vol. 1 1-203
sp_changegroup and the Vol. 1 1-205
sysreferences table **B-47** to **B-48**
sysremotelogins table **B-49**, Vol. 2 1-25 to 1-27, Vol. 2 1-151, Vol. 2 1-193
sp_dropremotelogin and Vol. 2 1-146
sysroles table **B-50**
syssegments table **B-51**, Vol. 2 1-149
sysservers table **B-52**
Backup Server and Vol. 1 1-169, Vol. 1 1-183
load database and Vol. 1 1-235
sp_addserver and Vol. 2 1-31
sp_helpremotelogins and Vol. 2 1-194
sp_helpserver and Vol. 2 1-200
syssrvroles table **B-53**
System activities
auditing Vol. 1 3-3 to 3-5
setting query-processing options for Vol. 1 1-293 to 1-305
shutdown Vol. 1 1-308
System Administrator Vol. 1 3-99
assigning role Vol. 2 1-256
System databases
dumping Vol. 1 1-168
System datatypes. *See* Datatypes
System functions **Vol. 1 2-40** to **2-47**
System logical name. *See* Logical device name
System messages, language setting for Vol. 1 1-297
See also Error messages; Messages
System procedures
See also **create procedure** command; Stored procedures; *individual procedure names*
catalog stored Vol. 2 2-1 to 2-36
changing names of Vol. 2 1-92
create procedure and Vol. 1 1-58 to 1-66
displaying syntax of Vol. 2 1-265 to 1-267
displaying the text of Vol. 2 1-204
dropping user-defined Vol. 1 1-152 to 1-153
help reports Vol. 2 1-171 to 1-209
isolation levels Vol. 1 3-137
list of Vol. 2 1-1 to 1-6
for login management Vol. 1 3-85
not allowed in user-defined transactions Vol. 1 3-130
permissions Vol. 2 1-6
on temporary tables Vol. 1 3-118
updating and B-3
using Vol. 2 1-7
System procedures results. *See* Information (Server)
System Security Officer Vol. 1 3-99
assigning role Vol. 2 1-256
system segment
alter database Vol. 1 1-9
dropping Vol. 2 1-149
mapping Vol. 2 1-29
System tables **B-1** to **B-3**
See also Tables; *individual table names*
affected by **drop table** Vol. 1 1-156
affected by **drop view** Vol. 1 1-160
allow updates variable and B-3
changes dangerous to B-3
changing reserved words Vol. 2 1-95
dbcc checkcatalog and Vol. 1 1-112
default definitions in Vol. 1 1-47
descriptions of individual B-5 to B-60
direct updates dangerous to Vol. 2 1-95
direct updates to B-3
fixing allocation errors found in Vol. 1 1-112
keys for B-29 to B-30
master database B-2
permissions on B-3
rebuilding of Vol. 1 1-112
rule information in Vol. 1 1-70
space allocation Vol. 2 1-233
sysname datatype Vol. 1 3-40
system procedures and Vol. 2 1-8
updating B-3, Vol. 2 1-1

systhresholds table **B-54**, Vol. 2 1-154, Vol. 2 1-206
systypes table **B-55** to **B-56**, Vol. 2 1-155
sysusages table **B-57**
sysusermessages table **B-58**
error message text Vol. 2 1-169
raiserror and Vol. 1 1-256
sp_dropmessage and Vol. 2 1-144
sysusers table **B-59** to **B-60**
sysalternates table and B-5, Vol. 2 1-9

T

Table columns. *See* Columns
Table locks
types of Vol. 2 1-213
Table pages
See also Pages (data)
allocation with **dbcc tablealloc** Vol. 1 1-111
system functions Vol. 1 2-41, Vol. 1 2-43
Table rows. *See* Rows, table
tablealloc option, **dbcc** Vol. 1 1-111
Tables
See also Database objects; System tables; *tempdb* database; Temporary tables
allowed in a **from** clause Vol. 1 1-283
auditing use of Vol. 2 1-55
changing names of Vol. 2 1-91
checking name with **sp_checkreswords** Vol. 2 1-88
column information Vol. 2 2-8 to 2-9
column permission information from **sp_column_privileges** Vol. 2 2-5 to 2-6
common key between Vol. 2 1-102 to 1-104
constraint information Vol. 2 1-175
creating duplicate Vol. 1 1-290
creating new Vol. 1 1-75 to 1-93, Vol. 1 1-282
creating with **create schema** Vol. 1 1-72 to 1-74
dbcc checkdb and Vol. 1 1-111
dividing, with **group by** and **having** clauses Vol. 1 1-207 to 1-216
dropping Vol. 1 1-156 to 1-158
dropping keys between Vol. 2 1-137
estimating space for Vol. 2 1-159
identifying Vol. 1 3-63
index location Vol. 1 1-150, Vol. 1 1-324
inner Vol. 1 3-82
isnull system function and Vol. 1 3-94
joined common key Vol. 2 1-102 to 1-104
joins of Vol. 1 3-79 to 3-84
locks held on Vol. 2 1-213
migration to a clustered index Vol. 1 1-55, Vol. 1 1-84
names as qualifiers Vol. 1 3-63
with no data Vol. 1 1-290
number allowed in a **from** clause Vol. 1 3-79
Object Allocation Maps of Vol. 1 1-111
object dependencies and B-22, Vol. 2 1-124 to 1-126
permissions on Vol. 1 1-197, Vol. 1 1-270
primary keys on Vol. 2 1-235, Vol. 2 2-17 to 2-18
renaming Vol. 2 1-248 to 1-250
single-group Vol. 1 1-210
sp_placeobject space allocation for Vol. 2 1-233 to 1-234
sp_recompile and Vol. 2 1-240 to 1-241
sp_table_privileges information on Vol. 2 2-32
sp_tables Vol. 2 2-35
space used by Vol. 2 1-263
splitting Vol. 2 1-233
with suspect indexes Vol. 2 1-210
system tables entries for B-13, B-40 to B-41
Transact-SQL extension effects and querying Vol. 1 1-211
update statistics on Vol. 1 1-324

work Vol. 1 2-4
Tables, temporary. *See tempdb* database; Temporary tables
tan mathematical function **Vol. 1 2-26**
Tangents, mathematical functions for Vol. 1 2-25 to 2-26
Tape dump devices
adding Vol. 2 1-46 to 1-48
sp_volchanged messages and Vol. 2 1-283
sysdevices table B-23
tape option, **sp_addumpdevice** Vol. 2 1-46
Technical Support Vol. 1 xxii, Vol. 2 xx
tempdb database
See also Databases
adding objects to Vol. 1 1-83
sysobjects table and Vol. 1 1-76
system tables entries and B-40 to B-41
systypes table and Vol. 1 1-83
user-defined datatypes in Vol. 1 3-41
Temporary names. *See* Aliases
Temporary tables **Vol. 1 3-116** to **3-119**
See also Tables; *tempdb* database
catalog stored procedures and Vol. 2 2-2
create procedure and Vol. 1 1-65
create table and Vol. 1 1-75, Vol. 1 1-83
identifier prefix (#) Vol. 1 1-75, Vol. 1 3-117
indexes on Vol. 1 1-55
names of Vol. 1 1-75, Vol. 1 3-116
naming Vol. 1 3-61
select into and Vol. 1 3-118
sp_help and Vol. 2 1-173
system procedures and Vol. 2 1-8
Terminals
7-bit, **sp_helpsort** output example Vol. 2 1-202
8-bit, **sp_helpsort** output example Vol. 2 1-203
Text
comment Vol. 1 3-10 to 3-11
comment, as control-of-flow language Vol. 1 3-12
copying with **defncopy** Vol. 2 1-90
user-defined message Vol. 2 1-23
text datatype **Vol. 1 3-120** to **3-124**
convert command Vol. 1 3-123
converting Vol. 1 2-14
initializing Vol. 1 3-120
initializing with null values Vol. 1 3-93
initializing with **update** Vol. 1 1-320
null values Vol. 1 3-121
separate storage of Vol. 1 1-261
sysindexes table and B-26
textsize setting Vol. 1 1-299
Text functions **Vol. 1 2-48** to **2-50**
Text page pointer Vol. 1 2-46
Text pointer values Vol. 1 2-48, Vol. 1 2-49
readtext and Vol. 1 1-261
textptr function Vol. 1 1-261, Vol. 1 1-262
textptr string function
text/image function Vol. 1 2-48
@@textsize global variable Vol. 1 1-299, Vol. 1 3-123, Vol. 1 3-148
readtext and Vol. 1 1-262
textsize option, **set** Vol. 1 1-299
textvalid string function
text/image function Vol. 1 2-48
Theta joins Vol. 1 3-80
@@thresh_hysteresis global variable Vol. 1 3-148
Threshold procedures
executing Vol. 2 1-36 to 1-37, Vol. 2 1-225
parameters passed to Vol. 2 1-36, Vol. 2 1-224
Thresholds
adding Vol. 2 1-34 to 1-39
changing Vol. 2 1-222 to 1-226
database dumps and Vol. 1 1-168
defining default procedure Vol. 2 1-268
disabling Vol. 2 1-37, Vol. 2 1-153, Vol. 2 1-225

last-chance Vol. 1 2-42, Vol. 2 1-35, Vol. 2 1-37, Vol. 2 1-153, Vol. 2 1-223, Vol. 2 1-225
maximum number Vol. 2 1-36, Vol. 2 1-224
procedures executed by Vol. 2 1-35, Vol. 2 1-223
removing Vol. 2 1-153 to 1-154
sp_helpthreshold information on Vol. 2 1-206
space between Vol. 2 1-36
systhresholds table B-54
transaction log dumps and Vol. 1 1-182
Ties, rules for sort order Vol. 1 1-250 to 1-251
Time interval
See also Timing
automatic checkpoint Vol. 1 1-25
elapsed execution (**statistics time**) Vol. 1 1-299
estimating index creation Vol. 2 1-159
for running a trigger Vol. 1 1-98
since **sp_monitor** last run Vol. 2 1-227
waitfor Vol. 1 1-327
time option, **waitfor** Vol. 1 1-327
Time values
datatypes Vol. 1 3-34 to 3-38
timeouts option
sp_serveroption Vol. 2 1-258
timestamp columns
automatic update of Vol. 1 3-39
creating Vol. 1 3-39
timestamp datatype Vol. 1 3-39 to 3-40
browse mode and Vol. 1 3-8, Vol. 1 3-39
comparison using **tsequal** function Vol. 1 2-43
Timestamps, order of transaction log dumps Vol. 1 1-234
@@timeticks global variable Vol. 1 3-148
Timing
See also Time interval
automatic checkpoint Vol. 1 1-25
@@error status check Vol. 1 3-146
tinyint datatype **Vol. 1 3-27**
titleauthor table, *pubs2* database C-6
titles table, *pubs2* database C-3 to C-5
to option
revoke Vol. 1 1-273
Topics list, Transact-SQL reference pages Vol. 1 3-1 to 3-2
@@total_errors global variable Vol. 1 3-148, Vol. 2 1-228
@@total_read global variable Vol. 1 3-148
sp_monitor and Vol. 2 1-228
@@total_write global variable Vol. 1 3-148
sp_monitor and Vol. 2 1-228
Totals
compute command Vol. 1 1-249
Trailing blanks. *See* Blanks
Trailing zeroes. *See* Zeroes
@@tranchained global variable Vol. 1 3-132, Vol. 1 3-148
@@trancount global variable Vol. 1 3-127, Vol. 1 3-148
Transaction canceling. *See* **rollback** command
transaction isolation level option
set Vol. 1 1-299
Transaction logs
See also **dump transaction** command; *syslogs* table
backing up Vol. 1 1-162
of deleted rows Vol. 1 1-127
dump database and Vol. 1 1-162
dumping Vol. 1 1-174
inactive space Vol. 1 1-175
insufficient space Vol. 1 1-181
load database and Vol. 1 1-235
loading Vol. 1 1-238 to 1-245
master database Vol. 1 1-168, Vol. 1 1-181
placing on separate segment Vol. 1 1-181
purging Vol. 1 1-169

on a separate device Vol. 1 1-132, Vol. 1 1-135, Vol. 1 1-141, Vol. 1 1-180, Vol. 1 3-47, Vol. 2 1-217 to 1-219
size Vol. 1 2-46
space extension Vol. 1 1-9
space, monitoring Vol. 1 1-182
syslogs table **trunc log on chkpt** Vol. 1 1-180
system tables entries for B-40 to B-41
thresholds and Vol. 2 1-153
transfer Vol. 1 1-180
writetext with log and Vol. 1 1-338
Transactions **Vol. 1 3-125** to **3-141**
See also Batch processing; **rollback** command; User-defined transactions
ANSI compliance Vol. 1 3-125
begin Vol. 1 1-21
canceling Vol. 1 3-128
chained Vol. 1 1-28, Vol. 1 3-131
cursors and Vol. 1 3-137
dump transaction command Vol. 1 1-174 to 1-186
ending with **commit** Vol. 1 1-28
and errors Vol. 1 3-139
fetch and Vol. 1 1-193
isolation levels Vol. 1 1-299
load transaction command Vol. 1 1-238 to 1-245
modes Vol. 1 3-131
modes, **sp_procxmode** Vol. 2 1-237 to 1-239
names not used in nested Vol. 1 3-131
nesting Vol. 1 3-127
number of databases allowed Vol. 1 3-128
parameters not part of Vol. 1 1-190
preparing Vol. 1 1-252
save transaction and Vol. 1 1-279 to 1-280
states Vol. 1 3-125
@@transtate global variable Vol. 1 3-125
unchained Vol. 1 3-131 to 3-132
update iteration within given Vol. 1 1-319
user-defined **Vol. 1 3-125** to **3-141**
Transact-SQL
aggregate functions in Vol. 1 2-5
commands summary table Vol. 1 1-1 to 1-5
extensions Vol. 1 1-211, Vol. 1 2-1
reserved words A-1 to A-2, Vol. 2 1-88
Transfer
command and object permissions Vol. 1 1-199
transaction log Vol. 1 1-180
Translation
of arguments Vol. 1 1-253
of integer arguments into binary numbers Vol. 1 3-55
of user-defined messages Vol. 2 1-23
@@transtate global variable Vol. 1 3-148
Trigger tables Vol. 1 1-99
Triggers
See also Database objects; Stored procedures
changing names of Vol. 2 1-92
checking name with **sp_checkreswords** Vol. 2 1-88
creating Vol. 1 1-94 to 1-102
delete and Vol. 1 1-127
displaying the text of Vol. 2 1-204
dropping Vol. 1 1-159
enabling self recursion Vol. 1 1-101
insert and Vol. 1 1-222
nested Vol. 1 1-100 to 1-101
nested, and **rollback trigger** Vol. 1 1-277
@@nestlevel and Vol. 1 1-101
object dependencies and B-22, Vol. 2 1-124 to 1-126
parseonly not used with Vol. 1 1-298
recursion Vol. 1 1-101
remapping Vol. 2 1-242 to 1-244
renamed database and Vol. 2 1-252
renaming Vol. 1 1-98, Vol. 2 1-248 to 1-250

rollback in Vol. 1 1-99, Vol. 1 1-276, Vol. 1 3-128
rolling back Vol. 1 1-277
@@rowcount and Vol. 1 1-99
self recursion Vol. 1 1-101
set commands in Vol. 1 1-293
sp_recompile and Vol. 2 1-240 to 1-241
stored procedures and Vol. 1 1-101
system tables entries for B-15, B-40 to B-41, B-42
time interval Vol. 1 1-98
transaction mode and Vol. 1 3-131
transactions and Vol. 1 3-134 to 3-139
truncate table command and Vol. 1 1-311
update and Vol. 1 1-319
Trigonometric functions
cosine Vol. 1 2-25
cotangent Vol. 1 2-25
sine Vol. 1 2-26
tangent Vol. 1 2-26
true | false clauses
sp_dboption Vol. 2 1-115
sp_remoteoption Vol. 2 1-245
sp_serveroption Vol. 2 1-258
true option, **sp_changedbowner** Vol. 2 1-79
True / false data, *bit* columns for Vol. 1 3-39
trunc log on chkpt option
setting with **sp_dboption** Vol. 2 1-119
truncate table command **Vol. 1 1-311** to **1-312**
auditing use of Vol. 2 1-52
delete triggers and Vol. 1 1-98
faster than **delete** command Vol. 1 1-126
update statistics after Vol. 1 1-324
Truncation
See also Padding, data
binary datatypes Vol. 1 3-32
character datatypes Vol. 1 3-29
datediff results Vol. 1 2-21
errors and datatypes Vol. 1 3-45
insert and Vol. 1 1-221
set string_rtruncation and Vol. 1 1-299
temporary table names Vol. 1 3-61, Vol. 1 3-116
trusted option, **sp_remoteoption** Vol. 2 1-245
Trusted status Vol. 2 1-26
Truth tables
bitwise operations Vol. 1 3-54
logical expressions Vol. 1 3-58 to 3-59
tsequal system function Vol. 1 2-43, Vol. 1 3-9
Twenty-first-century numbers Vol. 1 3-34
Two Phase Commit Probe Process B-36
Two-digit year numbers Vol. 1 2-20

U

Unbinding
defaults Vol. 1 1-47, Vol. 1 1-148, Vol. 2 1-271 to 1-273
rules Vol. 1 1-154
Unchained transaction mode Vol. 1 3-131 to 3-132
Unconditional branching to a user-defined label Vol. 1 1-195
Underscore (_)
character string wildcard Vol. 1 3-57, Vol. 1 3-106, Vol. 1 3-151
object identifier prefix Vol. 1 3-61
in temporary table names Vol. 1 3-61, Vol. 1 3-116
Undoing changes. *See* **rollback** command
union operator **Vol. 1 1-313** to **1-316**
cursors and Vol. 1 3-20
Unique constraints Vol. 1 1-87
unique keyword
alter table Vol. 1 1-12
create index Vol. 1 1-50
create table Vol. 1 1-77
Unique names as identifiers Vol. 1 3-62
Unlocking login accounts Vol. 2 1-215
Unmapping a segment from a database Vol. 2 1-148 to 1-150
Unmirroring devices. *See* Disk mirroring

Updatable cursors Vol. 1 1-122
update command **Vol. 1 1-317** to **1-323**
auditing use of Vol. 2 1-58
cursors and Vol. 1 3-19
ignore_dup_key and Vol. 1 1-52
ignore_dup_row and Vol. 1 1-53
insert and Vol. 1 1-220
null values and Vol. 1 3-91, Vol. 1 3-92, Vol. 1 3-93
triggers and Vol. 1 1-97, Vol. 1 1-100
views and Vol. 1 1-107, Vol. 1 1-322, Vol. 1 3-83
Update locks Vol. 2 1-213
in cursors Vol. 1 3-22
update statistics command **Vol. 1 1-324** to **1-325**
Updating
See also Changing; *timestamp* datatype
cursor rows Vol. 1 3-19
data in views Vol. 1 1-106, Vol. 1 1-107
direct to system tables B-3
"dirty" pages Vol. 1 1-25 to 1-26
ignore_dup_key and Vol. 1 1-52
prevention during browse mode Vol. 1 2-43
system procedures and B-3
system tables B-3
trigger firing per Vol. 1 1-101
while in browse mode Vol. 1 2-43, Vol. 1 3-8 to 3-9
writetext Vol. 1 1-338
upper string function Vol. 1 2-36
Uppercase letter preference Vol. 1 1-250
See also Case sensitivity; **order by** clause
us_english language B-31, Vol. 2 1-16
weekdays setting Vol. 1 2-22
use command **Vol. 1 1-326**
auditing use of Vol. 2 1-52
used_pgs system function Vol. 1 2-43, Vol. 1 2-46
user connections configuration variable Vol. 2 1-109
User errors. *See* Errors; Severity levels
User groups. *See* Groups
User groups. *See* Groups; "public" group
User IDs
displaying Vol. 2 1-129
dropping with **sp_droplogin** and Vol. 2 1-142
number 1, Database Owner Vol. 1 2-46
user_id function for Vol. 1 2-44
valid_user function Vol. 1 2-44
user keyword
alter table Vol. 1 1-11
create table Vol. 1 1-77
User names Vol. 1 2-44
See also Database object owners; Logins
changing Vol. 2 1-94
checking with **sp_checkreswords** Vol. 2 1-89
finding Vol. 1 2-43
User objects. *See* Database objects
User permissions. *See* Database Owners; Permissions
user system function Vol. 1 2-43
user_id system function Vol. 1 2-44
user_name system function Vol. 1 2-44, Vol. 1 2-46
User-created objects. *See* Database objects
User-defined datatypes **Vol. 1 3-40** to **3-41**
See also Datatypes
adding Vol. 2 1-40
adding with **sp_addtype** Vol. 2 1-40 to 1-45
binding defaults to Vol. 2 1-68 to 1-71
binding rules to Vol. 2 1-75
changing names of Vol. 2 1-92
checking name with **sp_checkreswords** Vol. 2 1-88
dropping Vol. 1 3-40, Vol. 2 1-155 to 1-156
IDENTITY columns and Vol. 1 3-74
naming Vol. 2 1-42
sysname as Vol. 1 3-40
temporary tables and Vol. 1 3-118
timestamp as Vol. 1 3-39

unbinding defaults from Vol. 2 1-271 to 1-273
unbinding rules with **sp_unbindrule** Vol. 2 1-276 to 1-278
User-defined messages Vol. 2 1-23 to 1-24
unbinding with **sp_unbindmsg** Vol. 2 1-274 to 1-275
User-defined stored procedures, executing Vol. 1 1-187 to 1-191
User-defined transactions **Vol. 1 3-125** to **3-141**
See also Transactions
begin transaction Vol. 1 1-21
ending with **commit** Vol. 1 1-28
Users
See also Aliases; Groups; Logins
accounting statistics Vol. 2 1-100, Vol. 2 1-255
adding Vol. 2 1-20 to 1-22
adding to a database with **sp_adduser** Vol. 2 1-49 to 1-51
auditing Vol. 1 3-3
change group for Vol. 2 1-81 to 1-82
changing account names with **sp_modifylogin** Vol. 2 1-220 to 1-221
changing names of Vol. 2 1-96
command and object permissions Vol. 1 1-202
dropping aliased Vol. 2 1-131 to 1-132
dropping from databases Vol. 2 1-157 to 1-158
dropping remote with **sp_dropserver** Vol. 2 1-151
dropping with **sp_droplogin** Vol. 2 1-142 to 1-143
guest Vol. 1 1-205, B-59, Vol. 2 1-158
impersonating (**setuser**) Vol. 1 1-199
information on Vol. 2 1-129, Vol. 2 1-208 to 1-209
information on permissions of Vol. 2 1-195
information on remote Vol. 2 1-193
logins information Vol. 2 1-193
management Vol. 1 3-85 to 3-87
other object owner Vol. 1 3-64
password change Vol. 2 1-230 to 1-232
remote, dropping with **sp_dropserver** Vol. 2 1-151
sp_who report on Vol. 2 1-286 to 1-287
sysloginroles table B-35
syslogins table B-36 to B-37
system procedure permissions and Vol. 2 1-6
system tables entries for B-36 to B-37, B-59
sysusers table B-59, Vol. 2 1-9
using bytes option, **patindex** string function Vol. 1 2-35
using option, **readtext** Vol. 1 1-261, Vol. 1 1-262
Utility commands
display syntax Vol. 2 1-265 to 1-267

V

valid_name system function Vol. 1 2-44, Vol. 1 3-65
valid_user system function Vol. 1 2-44
Values
configuration variable Vol. 1 1-264, Vol. 2 1-106 to 1-109
displaying with **sp_server_info** Vol. 2 2-19 to 2-22
dynamic and static Vol. 1 1-264
IDENTITY columns Vol. 1 1-223
procedure parameter or argument Vol. 1 1-188
system-generated Vol. 1 3-67
values option, **insert** Vol. 1 1-219
varbinary datatype **Vol. 1 3-32** to **3-33**
in *timestamp* columns Vol. 1 3-39
varchar datatype **Vol. 1 3-29** to **3-30**
datetime values conversion to Vol. 1 3-38
in expressions Vol. 1 3-59
spaces in Vol. 1 3-29
spaces in and **insert** Vol. 1 1-221

Variable-length character. *See varchar* datatype
Variable-length columns
null values in Vol. 1 3-89
offset B-13
stored order of Vol. 1 1-250
Variables **Vol. 1 3-143** to **3-149**
configuration Vol. 1 1-264
global Vol. 1 3-143 to 3-149
local Vol. 1 1-117 to 1-118, Vol. 1 3-143 to 3-149
passed as parameters Vol. 1 3-143
in **print** messages Vol. 1 1-254
return values and Vol. 1 1-189
sum or average integer data and Vol. 1 2-31
vdevno option
disk init Vol. 1 1-130
disk reinit Vol. 1 1-138
Vector aggregates
group by and Vol. 1 1-209
nesting Vol. 1 2-5
@@version global variable Vol. 1 1-254, Vol. 1 3-149
Views
See also Database objects; Multi-table views
allowed in a **from** clause Vol. 1 1-283
auditing use of Vol. 2 1-55
changes to underlying tables of Vol. 1 1-106
changing names of Vol. 2 1-92
check option and Vol. 1 1-321 to 1-322
checking name with **sp_checkreswords** Vol. 2 1-88
columns Vol. 2 2-8 to 2-9
common key between Vol. 2 1-102 to 1-104
creating Vol. 1 1-103 to 1-109
creating with **create schema** Vol. 1 1-72 to 1-74
displaying the text of Vol. 2 1-204
dropping Vol. 1 1-160
dropping keys between Vol. 2 1-137
IDENTITY columns and Vol. 1 3-73 to 3-74
inserting data through Vol. 1 1-224
joins and Vol. 1 3-79 to 3-84
names as qualifiers Vol. 1 3-63
number allowed in a **from** clause Vol. 1 3-79
object dependencies and B-22, Vol. 2 1-124 to 1-126
permissions on Vol. 1 1-197, Vol. 1 1-200, Vol. 1 1-270
primary keys on Vol. 2 1-235
pubs2 C-19
readtext and Vol. 1 1-263
remapping Vol. 2 1-242 to 1-244
renamed database and Vol. 2 1-252
renaming Vol. 1 1-106, Vol. 2 1-248 to 1-250
system tables entries for B-13, B-15, B-40 to B-41, B-42
update and Vol. 1 1-107, Vol. 1 1-321 to 1-322
updating restrictions Vol. 1 1-322
with check option Vol. 1 1-107, Vol. 1 1-224 to 1-225, Vol. 1 3-83
Violation of domain or integrity rules Vol. 1 1-221
Virtual address Vol. 1 1-138
Virtual device number Vol. 1 1-130, Vol. 1 1-133, Vol. 1 1-138
Virtual page numbers Vol. 2 1-181
Virtual tables B-3
Volume handling Vol. 2 1-279
vstart option
disk init Vol. 1 1-131
disk reinit Vol. 1 1-138

W

wait option, **shutdown** Vol. 1 1-308
waitfor command **Vol. 1 1-327** to **1-328**
Waiting for **shutdown** Vol. 1 1-309
week date part Vol. 1 2-20
weekday date part Vol. 1 2-20

Weekday date value
first Vol. 2 1-15
names and numbers Vol. 1 1-296, Vol. 1 2-22, Vol. 2 1-15
where clause **Vol. 1 1-329** to **1-335**
aggregate functions not permitted in Vol. 1 1-334
delete Vol. 1 1-125
difference from **having** clause Vol. 1 3-105
group by clause and Vol. 1 1-211
having and Vol. 1 1-334
joins and Vol. 1 3-80
null values in a Vol. 1 3-90
repeating a Vol. 1 1-215
where current of clause
delete Vol. 1 1-125
update Vol. 1 1-317
while keyword **Vol. 1 1-336** to **1-337**
while loop Vol. 1 1-336
continue Vol. 1 1-39
exit with **break** Vol. 1 1-23
Wildcard characters **Vol. 1 3-150** to **3-155**
See also **patindex** string function
ANSI SQL pattern matching ($ and _) Vol. 2 2-2
in expressions Vol. 1 3-57
in a **like** match string Vol. 1 1-188, Vol. 1 3-58
literal characters and Vol. 1 3-153
search conditions Vol. 1 3-106
used as literal characters Vol. 1 3-153
with check option option
create view Vol. 1 1-104
views and Vol. 1 1-108
with grant option option, **grant** Vol. 1 1-198
with keyword, **rollback trigger** Vol. 1 1-277
with log option, **writetext** Vol. 1 1-338
with no_error option, **set char_convert** Vol. 1 1-296
with nowait option, **shutdown** Vol. 1 1-308
with override option
alter database Vol. 1 1-7
for load and Vol. 1 1-42
reconfigure Vol. 1 1-264
with recompile option
create procedure Vol. 1 1-60
execute Vol. 1 1-188
with wait option, **shutdown** Vol. 1 1-308
wk. *See* **week** date part
Words, finding similar-sounding Vol. 1 2-38
Work session, **set** options for Vol. 1 1-293 to 1-305
Work tables, number of Vol. 1 2-4
Write operations, logging *text* or *image* Vol. 1 1-338
writes option, **disk mirror** Vol. 1 1-134, Vol. 1 3-48
writetext command **Vol. 1 1-338** to **1-340**
text data initialization requirement Vol. 1 3-122
triggers and Vol. 1 1-98

Y

year date part Vol. 1 2-20
Year values, date style Vol. 1 2-10
Yes / no data, *bit* columns for Vol. 1 3-39
yy. *See* **year** date part

Z

Zero x (0x) Vol. 1 2-10, Vol. 1 2-16, Vol. 1 3-32, Vol. 1 3-33
See also Binary datatypes; Hexadecimal numbers
Zeroes
leading or trailing Vol. 1 3-32
padding or inserting Vol. 1 3-32
trailing, in binary datatypes Vol. 1 3-32 to 3-33
using NULL or Vol. 1 3-88, Vol. 1 3-94
Zero-length string output Vol. 1 1-255